ATARI ST

GFA BASIC 3.0

Interpreter

Alle Rechte vorbehalten. Kein Teil dieses Handbuches darf ohne schriftliche Genehmigung der GFA Systemtechnik GmbH, in welcher Form und mit welchen Mitteln auch immer, sei es auf elektronischem oder mechanischem Weg, kopiert, vervielfältigt oder übermittelt werden. Dies gilt auch für Fotokopien und Tonbandaufnahmen, es sei denn, sie seien für den persönlichen Gebrauch des Käufers bestimmt.

Das auf der beiliegenden Diskette enthaltene Programm GFA-BASIC 3.0 Interpreter darf, außer zum persönlichen Gebrauch des Käufers, nicht kopiert werden.

Ausdrücklich gestattet ist die Weitergabe des mitgelieferten RUN-ONLY-Interpreters (GFABASRO.PRG).

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu veröffentlichen. GFA Systemtechnik übernimmt keine Gewähr dafür, daß die beschriebenen Verfahren, Programme usw. funktionsfähig und frei von Schutzrechten Dritter sind.

Dieses Handbuch wurde mit Microsoft Word erstellt und auf einem Kyocera Laserdrucker ausgedruckt.

(C) Copyright 1986, 1987, 1988 GFA Systemtechnik GmbH

Programmautor: Frank Ostrowski

Handbuchautoren: Gottfried P. Engels
Markus C. Görgens
Frank Ostrowski

Änderungen des Textes vorbehalten, Inhalt ohne Gewähr.

1. Auflage, Mai 1988

Inhaltsverzeichnis

1 - Einleitung

Zum Konzept dieses Handbuchs ..... 1 - 3

Erste Bekanntschaft mit GFA-BASIC 3.0 ..... 1 - 7

Der Editor ..... 1 - 11

Grundsätzliches ..... 1 - 11
Der Cursor-Tasten-Block ..... 1 - 12
Der Zifferntastenblock ..... 1 - 15
Weitere Editiertasten auf dem Haupttastaturfeld ..... 1 - 16
Control-Kommandos ..... 1 - 16
Die Menüleiste und die Funktionstasten ..... 1 - 19

Spezielles ..... 1 - 30

DEFBIT, DEFBYT, DEFINT, DEFWRD, DEFFLT, DEFSTR ..... 1 - 30
DEFLIST ..... 1 - 32
$ ..... 1 - 34

2 - Variablen und Speicherverwaltung

Variablentypen ..... 2 - 1

!, |, &, %, #, $ ..... 2 - 1

Felder ..... 2 - 3

DIM, DIM? ..... 2 - 4
OPTION BASE ..... 2 - 5
ARRAYFILL ..... 2 - 6

Typumwandlung ..... 2 - 7

TYPE ..... 2 - 7
ASC(), CHR$(), ... 2 - 8
STR$() ..... 2 - 9

BIN$(), OCT$(), HEX$() ..... 2 - 10
VAL(), VAL?() ..... 2 - 11
CVx(), MKx$() ..... 2 - 12
CINT(), CFLOAT() ..... 2 - 14

Pointer-Operationen ..... 2 - 15

* ..... 2 - 15
PEEK, POKE, DPEEK, LPEEK, POKE ..... 2 - 17
DPOKE, LPOKE, SPOKE, SDPOKE, SLPOKE ..... 2 - 17
BYTE{}, CARD{}, INT{}, LONG{}, {}, FLOAT{} ..... 2 - 19
SINGLE {}, DOUBLE{}, CHAR{} ..... 2 - 19
VARPTR, V:, ARRPTR, * ..... 2 - 23
ABSOLUTE ..... 2 - 24

Löschen und Vertauschen ..... 2 - 25

CLEAR, CLR, ERASE ..... 2 - 25
SWAP ..... 2 - 27
SSORT, QSORT ..... 2 - 29
INSERT, DELETE ..... 2 - 32

Reservierte Variablen ..... 2 - 34

FALSE, TRUE, PI ..... 2 - 34
DATE$, TIME$, TIMER, SETTIME, DATE$=, TIME$= ..... 2 - 35
TIMER ..... 2 - 36

Spezielles ..... 2 - 37

LET ..... 2 - 37
VOID, ~ ..... 2 - 38

Speicherverwaltung ..... 2 - 39

FRE ..... 2 - 39
BMOVE ..... 2 - 40
BASEPAGE, HIMEM ..... 2 - 41
RESERVE ..... 2 - 43
INLINE ..... 2 - 44
MALLOC, MFREE, MSHRINK ..... 2 - 45

3 - Operatoren

Arithmetische Operatoren ..... **3 - 2**

+ - * / ^ DIV \ MOD ..... 3 - 2
+ - ..... 3 - 2

Logische Operatoren ..... **3 - 4**

NOT ..... 3 - 5
AND ..... 3 - 6
OR ..... 3 - 7
XOR ..... 3 - 8
IMP ..... 3 - 10
EQV ..... 3 - 11

Zeichenkettenoperator ..... **3 - 12**

+ ..... 3 - 12

Vergleichsoperatoren ..... **3 - 13**

= == >= <= <> ..... 3 - 13

Zuweisungsoperator ..... **3 - 18**

= ..... 3 - 18

Operatorenhierarchie ..... **3 - 19**

() ..... 3 - 19

4 - Numerische Funktionen

Mathematische Funktionen ..... **4 - 1**

ABS, SGN ..... 4 - 2
ODD, EVEN ..... 4 - 3
INT, TRUNC, FIX, FRAC ..... 4 - 4
ROUND ..... 4 - 5
MAX, MIN ..... 4 - 6
SQR ..... 4 - 7
EXP, LOG, LOG10 ..... 4 - 8
SIN, COS, TAN, ASIN, ACOS, ATN, DEG, RAD, SINQ, COSQ ..... 4 - 9

Zufallszahlengenerator ..... **4 - 11**
RND, RANDOM, RAND, RANDOMIZE ..... 4 - 11

- Integer-Arithmetik

Befehle und Funktionen ..... **4 - 13**
DEC, INC ..... 4 - 14
ADD, SUB, MUL, DIV ..... 4 - 15
PRED(), SUCC() ..... 4 - 16
ADD(), SUB(), MUL(), DIV(), MOD() ..... 4 - 17

Bit-Operationen ..... **4 - 19**
BCLR, BSET, BCHG, BTST ..... 4 - 20
SHL, SHR, ROL, ROR ..... 4 - 22
AND(), OR(), XOR(), IMP(), EQV() ..... 4 - 24
SWAP() ..... 4 - 26
BYTE(), CARD(), WORD () ..... 4 - 27

5 - Zeichenkettenverwaltung

LEFT$, RIGHT$ ..... 5 - 2
MID$ (als Funktion) ..... 5 - 3
PRED, SUCC ..... 5 - 4
LEN, TRIM$ ..... 5 - 5
INSTR ..... 5 - 6
RINSTR ..... 5 - 7
STRING$, SPACE$, SPC ..... 5 - 8
UPPER$ ..... 5 - 9
LSET, RSET, MID$ (als Befehl) ..... 5 - 10

6 - Tastaturein- und Bildschirmausgabe

INKEY$ ..... 6 - 2
INPUT ..... 6 - 3
LINE INPUT ..... 6 - 5
FORM INPUT, FORM INPUT AS ..... 6 - 6

PRINT, WRITE, PRINT AT(), LOCATE ..... 6 - 7
PRINT USING, PRINT AT() USING ..... 6 - 10
MODE ..... 6 - 12
DEFNUM ..... 6 - 13
CRSCOL, CRSLIN, POS, TAB ..... 6 - 14
HTAB, VTAB ..... 6 - 14

Die KEYxxx-Befehle ..... 6 - 16

KEYPAD ..... 6 - 16
KEYTEST, KEYGET, KEYLOOK ..... 6 - 18
KEYPRESS ..... 6 - 21
KEYDEF ..... 6 - 22

- Allgemeine Ein- und Ausgabe

Datenzeilen ..... 6 - 26

DATA, READ, RESTORE ..... 6 - 26

Dateiverwaltung ..... 6 - 28

Verzeichnisse ..... 6 - 30
DFREE(), CHDRIVE, DIR$, CHDIR ..... 6 - 30
DIR, FILES ..... 6 - 32
FGETDTA, FSETDTA ..... 6 - 33
FSFIRST, FSNEXT ..... 6 - 34
MKDIR, RMDIR ..... 6 - 35

Dateien ..... 6 - 36
EXIST ..... 6 - 36
OPEN ..... 6 - 37
LOF(), LOC(), EOF(), CLOSE, TOUCH ..... 6 - 39

NAME AS, RENAME AS, KILL ..... 6 - 41
BLOAD, BSAVE, BGET, BPUT ..... 6 - 42

Sequentieller und indexsequentieller Zugriff ..... 6 - 44

INP#, OUT# ..... 6 - 44
INPUT$(#) ..... 6 - 45
INPUT#, LINE INPUT# ..... 6 - 46
PRINT#, PRINT# USING, WRITE# ..... 6 - 47
STORE, RECALL ..... 6 - 49
SEEK, RELSEEK ..... 6 - 51

Wahlfreier Zugriff ..... 6 - 53

FIELD AS, AT ..... 6 - 54
GET #, PUT #, RECORD ..... 6 - 55

Kommunikation mit der Peripherie ..... 6 - 57

Byteweise Ein- und Ausgabe ..... 6 - 57
INP(), INP?(), OUT, OUT#, OUT?() ..... 6 - 57

Serielle- und MIDI-Schnittstelle ..... 6 - 59
INPAUX$, INPMID$ ..... 6 - 59

Maus und Joystick ..... 6 - 60
MOUSEX, MOUSEY, MOUSEK, MOUSE ..... 6 - 60
SETMOUSE ..... 6 - 62
HIDEM, SHOWM ..... 6 - 63
STICK, STICK(), STRIG() ..... 6 - 64

Drucker ..... 6 - 66
LPRINT, LPOS(), HARDCOPY ..... 6 - 66

Tonerzeugung ..... 6 - 68
SOUND, WAVE ..... 6 - 68

7 - Programmsteuerung

Entscheidungsanweisungen ..... **7 - 3**

IF THEN ELSE ENDIF ..... 7 - 3
ELSE IF ..... 7 - 5

Mehrfachverzweigungen ..... **7 - 7**

ON GOSUB ..... 7 - 7
SELECT, CASE, DEFAULT, ENDSELECT, CONT ..... 7 - 8

Schleifen ..... **7 - 13**

FOR, STEP, NEXT ..... 7 - 14
REPEAT, UNTIL ..... 7 - 16
WHILE, WEND ..... 7 - 18
DO, LOOP ..... 7 - 19
DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL ..... 7 - 20
EXIT IF ..... 7 - 22

Prozeduren und Funktionen ..... **7 - 23**

GOSUB, @, PROCEDURE, RETURN ..... 7 - 24
LOCAL ..... 7 - 26

@funk, FUNCTION, RETURN x, ENDFUNC ..... 7 - 27
DEFFN, FN ..... 7 - 29

Ereignisbezogene Verzweigungen ..... **7 - 31**

ON BREAK, ON BREAK CONT, ON BREAK GOSUB ..... 7 - 32
ON ERROR, ON ERROR GOSUB, RESUME, RESUME NEXT ..... 7 - 33
ERROR, ERR, ERR$, FATAL ..... 7 - 35

Interrupt-Programmierung ..... **7 - 37**

EVERY, EVERY STOP, EVERY CONT ..... 7 - 37
AFTER, AFTER STOP, AFTER CONT ..... 7 - 37

Sonstige ..... **7 - 40**

REM, ', ! ..... 7 - 40
GOTO ..... 7 - 41
PAUSE, DELAY ..... 7 - 42

END, EDIT, STOP ..... 7 - 43
NEW ..... 7 - 44
LOAD ..... 7 - 45
SAVE, PSAVE ..... 7 - 46
LIST, LLIST ..... 7 - 47
CHAIN ..... 7 - 48
RUN ..... 7 - 49
SYSTEM, QUIT ..... 7 - 50

Fehlerbehandlung ..... 7 - 51

TRON, TRON#, TROFF ..... 7 - 51
TRON proc, TRACE$ ..... 7 - 53
DUMP ..... 7 - 55

8 - Grafik

Definitionsbefehle ..... 8 - 1

SETCOLOR, COLOR ..... 8 - 2
DEFMOUSE ..... 8 - 3
DEFMARK ..... 8 - 5
DEFFILL ..... 8 - 6
BOUNDARY ..... 8 - 9
DEFLINE ..... 8 - 10
DEFTEXT ..... 8 - 12
GRAPHMODE ..... 8 - 14

Allgemeine Grafikbefehle ..... 8 - 15

CLIP ..... 8 - 16
PLOT, LINE, DRAW ..... 8 - 17
DRAW, DRAW (), SETDRAW ..... 8 - 19
BOX, PBOX, RBOX, PRBOX ..... 8 - 23
CIRCLE, PCIRCLE, ELLIPSE, PELLIPSE ..... 8 - 24
POLYLINE, POLYMARK, POLYFILL ..... 8 - 25
POINT() ..... 8 - 26
FILL ..... 8 - 27
CLS ..... 8 - 28

TEXT ..... 8 - 29
SPRITE ..... 8 - 30

Bildschirmausschnitte ..... 8 - 32

SGET, SPUT ..... 8 - 32
GET, PUT ..... 8 - 33
VSYNC ..... 8 - 35
BITBLT ..... 8 - 36

9 - Ereignis-, Menü- und Fensterverwaltung

Ereignisverwaltung ..... 9 - 1

ON MENU ..... 9 - 2
MENU() ..... 9 - 3
ON MENU BUTTON GOSUB ..... 9 - 8
ON MENU KEY GOSUB ..... 9 - 10
ON MENU IBOX GOSUB, ON MENU OBOX GOSUB ..... 9 - 11
ON MENU MESSAGE GOSUB ..... 9 - 13

Pull-Down-Menüs ..... 9 - 15

ON MENU GOSUB, MENU m$() ..... 9 - 16
MENU OFF, MENU KILL ..... 9 - 18
MENU ..... 9 - 19

Fensterbefehle ..... 9 - 22

OPENW, CLOSEW ..... 9 - 23
W_HAND, W_INDEX ..... 9 - 25
CLEARW, TITLEW, INFOW, TOPW, FULLW ..... 9 - 26
WINDTAB ..... 9 - 27

Sonstige ..... 9 - 30

RC_INTERSECT ..... 9 - 30
RC_COPY TO ..... 9 - 32
ALERT ..... 9 - 33
FILESELECT ..... 9 - 35

10 - Systemroutinen

GEMDOS, BIOS, XBIOS ..... 10 - 1
L:, W: ..... 10 - 3

Line-A-Aufrufe ..... 10 - 4

ACLIP ..... 10 - 5
PSET ..... 10 - 6
PTST() ..... 10 - 7
ALINE ..... 10 - 8
HLINE ..... 10 - 9
ARECT ..... 10 - 10
APOLY TO ..... 10 - 11
BITBLT ..... 10 - 12
ACHAR ..... 10 - 17
ATEXT ..... 10 - 18
L~A ..... 10 - 19

VDI-Aufrufe ..... 10 - 20

CONTRL(), INTIN(), PTSIN(), INTOUT(), PTSOUT() ..... 10 - 22
VDISYS ..... 10 - 23
VDIBASE ..... 10 - 25
WORK_OUT() ..... 10 - 26

Spezielle VDI-Routinen und GDOS ..... 10 - 27

GDOS? ..... 10 - 29
V~H ..... 10 - 30
V_OPNWK, V_CLSWK ..... 10 - 31
V_OPNVWK, V_CLSVWK ..... 10 - 32
V_CLRWK, V_UPDWK ..... 10 - 33
VST_LOAD_FONTS, VST_UNLOAD_FONTS ..... 10 - 35
VQT_EXTENT ..... 10 - 37
VQT_NAME ..... 10 - 38

Fremdsprachen-Routinen-Aufruf ..... 10 - 39

C: ..... 10 - 39
MONITOR ..... 10 - 42
CALL ..... 10 - 43

RCALL ..... 10 - 44
EXEC ..... 10 - 46

11 - AES-Bibliotheken

GCONTRL, ADDRIN, ADDROUT, GINTIN, GINTOUT, GB ..... 11 - 2
GEMSYS ..... 11 - 3

Objekt-Struktur ..... 11 - 4

OB_NEXT, OB_HEAD, OB_TAIL ..... 11 - 4
OB_TYPE, OB_SPEC ..... 11 - 5
OB_STATE, OB_FLAGS, OB_X, OB_Y, OB_W, OB_H ..... 11 - 6
OB_ADR ..... 11 - 7
Textinformations-Struktur (TEDINFO) ..... 11 - 8
Icon-Struktur (ICONBLK) ..... 11 - 9
Bit-Image-Struktur (BITBLK) ..... 11 - 10
Applikations-Block-Struktur (USERBLK) ..... 11 - 11
Parameter-Block-Struktur (PARMBLK) ..... 11 - 12

Applikations-Bibliothek (application library) ..... 11 - 13

APPL_INIT ..... 11 - 13
APPL_READ ..... 11 - 14
APPL_WRITE ..... 11 - 15
APPL_FIND ..... 11 - 16
APPL_TPLAY ..... 11 - 17
APPL_TRECORD ..... 11 - 17
APPL_EXIT ..... 11 - 18

Ereignis-Bibliothek (event library) ..... 11 - 19

EVNT_KEYBD ..... 11 - 19
EVNT_BUTTON ..... 11 - 20
EVNT_MOUSE ..... 11 - 22
EVNT_MESAG ..... 11 - 24
EVNT_TIMER ..... 11 - 25
EVNT_MULTI ..... 11 - 26
EVNT_DCLICK ..... 11 - 27

Menü-Bibliothek (menu library) ..... 11 - 28

MENU_BAR ..... 11 - 28
MENU_ICHECK ..... 11 - 29
MENU_IENABLE ..... 11 - 30
MENU_TNORMAL ..... 11 - 31
MENU_TEXT ..... 11 - 32
MENU_REGISTER ..... 11 - 33

Objekt-Bibliothek (object library) ..... 11 - 34

OBJC_ADD ..... 11 - 34
OBJC_DELETE ..... 11 - 35
OBJC_DRAW ..... 11 - 36
OBJC_FIND ..... 11 - 37
OBJC_OFFSET ..... 11 - 38
OBJC_ORDER ..... 11 - 39
OBJC_EDIT ..... 11 - 40
OBJC_CHANGE ..... 11 - 41

Formular-Bibliothek (form library) ..... 11 - 42

FORM_DO ..... 11 - 42
FORM_DIAL ..... 11 - 43
FORM_ALERT ..... 11 - 44
FORM_ERROR ..... 11 - 45
FORM_CENTER ..... 11 - 46
FORM_KEYBD ..... 11 - 47
FORM_BUTTON ..... 11 - 48

Grafik-Bibliothek (graphics library) ..... 11 - 49

GRAF_RUBBERBOX ..... 11 - 49
GRAF_DRAGBOX ..... 11 - 51
GRAF_MOVEBOX ..... 11 - 52
GRAF_GROWBOX ..... 11 - 53
GRAF_SHRINKBOX ..... 11 - 54
GRAF_WATCHBOX ..... 11 - 55
GRAF_SLIDEBOX ..... 11 - 56
GRAF_HANDLE ..... 11 - 57
GRAF_MOUSE ..... 11 - 58
GRAF_MKSTATE ..... 11 - 59

Scrap-Bibliothek (srap library) ..... **11 - 60**

SCRP_READ ..... 11 - 60
SCRP_WRITE ..... 11 - 61

Dateiauswahl-Bibliothek (file selector library) ..... **11 - 62**

FSEL_INPUT ..... 11 - 62

Fenster-Bibliothek (window library) ..... **11 - 63**

WIND_CREATE ..... 11 - 63
WIND_OPEN ..... 11 - 64
WIND_CLOSE ..... 11 - 65
WIND_DELETE ..... 11 - 66
WIND_GET ..... 11 - 67
WIND_SET ..... 11 - 70
WIND_FIND ..... 11 - 72
WIND_UPDATE ..... 11 - 73
WIND_CALC ..... 11 - 74

Resource-Bibliothek (resource library) ..... **11 - 75**

RSRC_LOAD ..... 11 - 75
RSRC_FREE ..... 11 - 76
RSRC_GADDR ..... 11 - 77
RSRC_SADDR ..... 11 - 78
RSRC_OBFIX ..... 11 - 79

Shell-Bibliothek (shell library) ..... **11 - 80**

SHEL_READ ..... 11 - 80
SHEL_WRITE ..... 11 - 81
SHEL_GET ..... 11 - 82
SHEL_PUT ..... 11 - 83
SHEL_FIND ..... 11 - 84
SHEL_ENVRN ..... 11 - 85

Beispielprogramme ..... **11 - 86**

12 - Anhang

Kompatibilität zu GFA-BASIC 2.xx ..... **12 - 1**

GEMDOS-Tabelle ..... 12 - 4
BIOS-Tabelle ..... 12 - 15
XBIOS-Tabelle ..... 12 - 18
Tabelle der Line-A-Variablen ..... 12 - 27
Tabelle der Eingabe-Parameter für V_OPN(v)WK ..... 12 - 30
VT 52-Tabelle ..... 12 - 33
Scan-Code-Tabelle ..... 12 - 34
ASCII-Tabelle ..... 12 - 35
Spezielle ASCII-Zeichen ..... 12 - 36
Füllmuster-Tabelle ..... 12 - 37
Linienstil-Tabelle ..... 12 - 37

Fehlermeldungen ..... **12 - 39**

GFA-BASIC-Fehlermeldungen ..... 12 - 39
Bomben-Fehlermeldungen ..... 12 - 41
TOS-Fehlermeldungen ..... 12 - 41
Editor-Fehlermeldungen ..... 12 - 43

Index ..... **12 - 45**

1 - Einleitung

Mit dem GFA-BASIC-Interpreter 3.0 steht Ihnen eine umfangreiche Programmiersprache mit einer komfortablen Entwicklungsumgebung zur Verfügung. Dieser fortschrittliche BASIC-Dialekt verfügt über einen leistungsfähigen und schnellen Editor, der die Erstellung von strukturierten Programmen unterstützt.

Der Interpreter bietet eine komfortable Möglichkeit zur Fehlerbehandlung durch die besonderen Kommandos zur Fehlersuche und Fehlerbehebung (z.B. Tron procedure und TRACE$). In modernen Programmiersprachen sind die Möglichkeiten der Programmstrukturierung von besonderer Bedeutung. Bereits der Editor unterstützt die strukturierte Programmierung durch automatische Einrückung von Befehlen in Schleifen und Bedingungen. Darüber hinaus können Unterprogramme im Programm-Listing durch einen Platzhalter vertreten sein, an dessen Stelle durch einen Tastendruck das Unterprogramm "aufgeklappt" werden kann.

Im Bereich der bedingten Anweisungen sind zu den Befehlen IF-ELSE-ENDIF, die bereits in älteren Versionen von GFA-BASIC existierten, noch Befehle zur Mehrfachverzweigung hinzugekommen (ELSE IF, SELECT-CASE).

Um Unterprogramme formulieren zu können, gibt es die Möglichkeit, Prozeduren und Funktionen (neu in 3.0) zu vereinbaren. Dabei können sowohl Variablenwerte als auch die Variablen selbst übergeben werden. Ferner stehen neben den bereits aus älteren GFA-BASIC-Versionen bekannten Schleifentypen FOR-NEXT, REPEAT-UNTIL, WHILE-WEND und DO-LOOP erweiterte Schleifenkommandos wie DO-UNTIL, DO-WHILE, LOOP-UNTIL und LOOP-WHILE zur Verfügung.

Systemnahe Programmierung wird durch die Möglichkeit des Aufrufs von Betriebssystemroutinen (GEMDOS, BIOS, XBIOS) gestattet. Viele dieser Funktionen stehen auch in Form einfacher Befehle zur Verfügung. Interruptgesteuerte Programmierung mit EVERY und AFTER sind ebenfalls

möglich. Assembler- und C-Unterprogramme lassen sich mit Befehlen wie z.B. RCALL, C: und Monitor einbinden.

Die wichtigsten VDI-Routinen und alle AES-Funktionen (z.B. Menü-, Fenster- und Formularverwaltungsfunktionen) sind als eingebaute Funktionen verfügbar. Dies ermöglicht zusammen mit zusätzlichen, besonders einfach gehaltenen Befehlen zur GEM-Einbindung eine leichte Entwicklung von Programmen, die eine GEM-Benutzeroberfläche haben sollen.

Der GFA-BASIC-Interpreter 3.0 verfügt über echte Integer-Arithmetik, die eine hohe Rechengeschwindigkeit bietet, sowie über Fließkomma-Arithmetik mit hoher Rechengenauigkeit (13 Nachkommastellen).

Gegenüber den 2.xx-Versionen des GFA-BASIC sind weitere Variablentypen (BYTE, WORD) und Bit-Operationen (BCLR, BSET, BTST, BCHG, SHL, SHR, ROL, ROR usw.) hinzugekommen. Grafikprogramme können einfachen Gebrauch von Line-A-Routinen machen, die als Befehle implementiert sind.

Zum Konzept dieses Handbuches

Dieses Handbuch beginnt mit einer kurzen Beschreibung der Programmiersprache GFA-BASIC 3.0. Danach folgen eine Erläuterung des Handbuchaufbaus (die Sie gerade lesen) und eine Einführung in die Benutzung des GFA-BASIC-Interpreters. Das Kapitel endet mit einer Beschreibung der Punkte, die man bei der Übernahme von Programmen, die in älteren GFA-BASIC-Versionen geschrieben wurden, beachten muß. Das nächste Kapitel beschreibt die Kommandos zur Bedienung des Editors. Alle danach folgenden Kapitel beinhalten im wesentlichen eine Beschreibung der Befehle und Funktionen von GFA-BASIC 3.0. Diese sind nach inhaltlichen Gesichtspunkten geordnet, mit gemeinsamer Erklärung verwandter Begriffe (z.B. MIN und MAX). Im Anhang findet sich eine alphabetische Übersicht der Befehle und Funktionen, mit Verweisen auf die entsprechenden Seiten der Handbuchseiten mit den Erläuterungen. Diese sind wie folgt aufgebaut:

- Syntaxangabe.
- Beschreibung der zulässigen Parametertypen.
- Erläuternder Text.
- Beispiel.

Bei der Syntaxangabe sind optionale Parameter mit eckigen Klammern markiert worden, z.B. so:

```
LEFT$(a$ [,x] )
```

In GFA-BASIC gibt es Befehle und Funktionen. Befehle geben keinen Wert zurück:

```
LINE 100,100,200,200
```

Funktionen liefern einen Rückgabewert. Diesen kann man mit PRINT ausgeben, einer Variablen zuweisen oder in einem Ausdruck verrechnen. Nachfolgend einige Beispiele:

```
PRINT ASC("65")
PRINT ASC("A")
a = ASC("A")
b = ASC("A") + 32
```

In diesem Handbuch wird bei der Syntaxangabe von Funktionen nicht verdeutlicht, daß sie einen Rückgabewert haben, sondern dies geht aus der Beschreibung der Funktionen hervor. Als Syntaxangabe wird dann z.B. nur ASC(a$) angegeben.

Wahlfreie Parameter, von denen bei einem Befehl beliebig viele angegeben werden können (z.B. bei DATA), werden ebenfalls mit eckigen Klammern aufgeführt. In diesen stehen zwei Parameter, gefolgt von drei Punkten, z.B.:

```
DATA [x,y,...]
```

Die Angabe der erlaubten Parametertypen steht unter der Syntaxangabe. Für diese Typen werden folgende Abkürzungen verwendet:

avar Arithmetische Variable. Dabei muß es sich um eine numerische Variable beliebigen Typs handeln.

aexp Arithmetischer Ausdruck (arithmetic expression). Dies ist ein beliebig komplexer Ausdruck, der eine Zahl produziert. Dabei kann es sich auch um eine Konstante (eine Zahl) handeln oder um eine Variable (Variablen sind eine Teilgruppe der Ausdrücke). Beispiele für arithmetische Ausdrücke sind:

```
a%
3
2+a%+ASC("A")
```

svar String-Variable. Dies ist eine Zeichenkettenvariable. Dieser Variablentyp hat die Endung $.

sexp Zeichenkettenausdruck (string expression). Dieser Ausdruck kann von beliebiger Komplexität sein und muß einen String ergeben. Dabei kann es sich auch um eine Konstante handeln (ein Text in Anführungszeichen) oder um eine Zeichenkettenvariable. Beispiele für String-Ausdrücke sind:

```
a$
"Test"
a$+"Probe"+LEFT$("HANDBUCH",4)
```

ivar Ganzzahlige Variable (integer variable).

iexp Ganzzahliger Ausdruck (integer expression).

bexp Logischer Ausdruck (boolean expression).

Wichtig ist, daß bei einigen numerischen Parametern nicht beliebige numerische Variablentypen angegeben werden können. Das wichtigste Beispiel hierfür sind Adressen. Adressen müssen mindestens in einer Vier-Byte-Variable angegeben werden; Boolean-, Byte- oder Word-Variablen sind also nicht zulässig. Nach der Beschreibung der zulässigen Parametertypen folgt die Erläuterung der Befehle. Dort wird die Bedeutung des Befehls und seiner einzelnen Parameter erklärt.

Die Besprechung eines Befehls wird mit einem oder mehreren Beispielen beendet. Jedes dieser Beispiele ist im Editor einzugeben und mit RUN (Shift + F10 oder Anklicken von RUN in der Menüleiste) zu starten. Hinter jedem Beispiel ist angegeben, welchen Effekt es produziert.

Dieses Konzept der Befehlsbeschreibung wurde nur im Abschnitt über die AES-Bibiliotheken etwas abgewandelt. Hier wird der Name des entsprechenden Befehls angegeben, danach folgt die Beschreibung seiner Funktion. Erst dann folgt die Syntax der Anweisung mit der Erklärung der einzelnen Parameter. Am Ende des Kapitels über die AES-Bibliotheken sind mehrere längere Beispielprogramme aufgeführt.

Ein Grund für diesen Aufbau ist, daß viele Befehle dieses Kapitels nur mit mehreren anderen Befehlen sinnvoll zusammenarbeiten, so daß Beispielprogramme erzwungenermaßen viele AES-Befehle enthalten und etwas umfangreicher sein müssen.

Das Handbuch schließt mit einer Sammlung von Tabellen und einer alphabetisch sortierten Liste aller Befehle, in der die Seitennummern angegeben sind, auf denen die Befehle beschrieben werden.

Erste Bekanntschaft mit GFA-BASIC 3.0

Dieser Abschnitt ist für diejenigen Anwender von GFA-BASIC 3.0 gedacht, die bisher noch keine Erfahrungen mit dieser Programmiersprache gemacht haben. Wer also schon mit den älteren Versionen gearbeitet hat, kann diesen Abschnitt überschlagen.

Die GFA-BASIC-Programmdiskette ist nicht kopiergeschützt. Erstellen Sie also als erstes eine Sicherheitskopie der Originaldiskette. Die Beschreibung des Kopiervorgangs finden Sie im Handbuch Ihres Rechners. Legen Sie dann die Kopie der Programmdiskette in Ihr Diskettenlaufwerk und starten Sie den GFA-BASIC-Interpreter.

Auf dem Bildschirm erscheint nun der Editor, in dem Sie Ihre Programme schreiben können. Geben Sie nun die folgenden Programmzeilen ein, und drücken Sie nach der Eingabe jeder Zeile die Return-Taste.

Dabei brauchen Sie die Leerzeichen, die vor einigen Befehlen stehen, nicht mit einzugeben. Die Einrückungen von Befehlen in Schleifen werden vom Editor automatisch vorgenommen. Es ist auch nicht nötig, bei der Eingabe der Befehlszeilen auf Groß- und Kleinschreibung zu achten. Wenn Sie eine Zeile mit Return verlassen, wird die Groß- und Kleinschreibung automatisch durchgeführt.

```
DEFFILL 1,0
REPEAT
  WHILE MOUSEK=1
    PBOX MOUSEX,MOUSEY,MOUSEX+30,MOUSEY+30
  WEND
UNTIL MOUSEK=2
```

In der rechten oberen Bildschirmecke finden Sie das Wort "Run". Zeigen Sie mit dem Mauspfeil auf dieses Wort und drücken Sie die linke Maustaste, um das Programm zu starten.

Es erscheint nun ein weißer Bildschirm, auf dem der Mauspfeil sichtbar ist. Wenn Sie nun die linke Maustaste drücken und die Maus über den Tisch bewegen, können Sie auf dem Bildschirm zeichnen. Als "Pinsel" wird ein Rechteck verwendet. Das Drücken der rechten Maustaste beendet das Programm. Es erscheint dann eine Alert-Box auf dem Bildschirm, die die Meldung "Programmende" enthält. Zeigen Sie auf das Wort "Return" in dieser Box und drücken Sie die linke Maustaste. Anschließend befinden Sie sich wieder im Editor.

Wie funktioniert dieses Programm? Das Zeichnen erfolgt mit dem PBOX-Befehl in der Mitte des Programms. Er zeichnet ausgefüllte Rechtecke. Seine vier Parameter geben die Eckpunkte dieses Rechtecks an. Der erste Befehl des Programms (DEFFILL 1,0) legt fest, daß das Rechteck im Inneren weiß ausgefüllt werden soll.

Die Variablen MOUSEK, MOUSEX und MOUSEY beinhalten Informationen über die Maus. In MOUSEK steht, welche Maustaste gerade gedrückt ist. Dabei bedeutet MOUSEK=1, daß die linke und MOUSEK=2, daß die rechte Maustaste gedrückt wird. In MOUSEX und MOUSEY steht die x- bzw. y-Position des Aktionspunktes des Maus-Cursors auf dem Bildschirm (vergl. Grafik, DEFMOUSE).

Die restlichen Befehle (REPEAT, WHILE, WEND, UNTIL) sind Schleifenbefehle. Die Schleife aus den Befehlen WHILE MOUSEK=1 und WEND läßt sich in die Formulierung "wiederhole, solange die linke Maustaste gedrückt ist" übersetzen. Die äußere Schleife aus den Befehlen REPEAT und UNTIL MOUSEK=2 bedeutet soviel wie: "wiederhole, bis die rechte Maustaste gedrückt wird". Da hinter der Zeile mit dem UNTIL-Befehl keine weitere Anweisung mehr folgt, wird das Programm nach dem Verlassen der REPEAT-UNTIL-Schleife beendet.

Gehen Sie nun hinter die letzte Zeile des Programms und geben Sie folgende, absichtlich falsche Programmzeile ein, in der das i des Befehlswortes PRINT fehlt:

```
prnt "Test"
```

Wenn Sie nun die Return-Taste drücken, um den Befehl zu bestätigen und die Zeile zu verlassen, so ertönt ein Signalton, und in der zweiten Bildschirmzeile erscheint die Meldung "Syntax Error".

Der Editor prüft also schon während der Programmerstellung, ob die Befehle syntaktisch korrekt eingegeben werden. Gehen Sie nun mit dem Cursor auf den Buchstaben r und drücken Sie dreimal die Delete-Taste. In der Programmzeile steht jetzt nur noch:

```
p "Test"
```

Wenn Sie nun die Return-Taste drücken, so können Sie die Zeile verlassen, und der Buchstabe p wird automatisch als Abkürzung für das Befehlswort PRINT erkannt.

Dies soll als erste Kontaktaufnahme mit dem Interpreter genügen.

Der Editor

Grundsätzliches

Der Editor des GFA-BASIC 3.0 ist kein gewöhnlicher Text-Editor, sondern speziell für die Entwicklung von Programmen konzipiert. Dies zeigt sich z.B. darin, daß syntaktisch falsche Befehle bereits während der Programmerstellung erkannt werden. Außerdem werden Befehle in Schleifen oder bedingten Anweisungen automatisch eingerückt und Befehlsabkürzungen in den vollständigen Befehlsnamen überführt (z.B. p in PRINT).

Beim Schreiben eines Programm-Listings wird immer dann, wenn eine Zeile verlassen werden soll, eine Syntaxkontrolle durchgeführt. Ist der Befehl in dieser Zeile syntaktisch nicht korrekt, so erscheint in der zweiten Zeile die Meldung "Syntax Error". Durch das Setzen des Remark-Zeichens (') am Zeilenanfang wird diese Zeile zum Kommentar. Danach ist es möglich, die Zeile zu verlassen.

In jede Programmzeile kann nur ein Befehl aufgenommen werden. Hinter diesem Befehl kann lediglich noch ein Remark-Text stehen. Dieser eine Befehl darf nicht mehr als 255 Zeichen lang sein. Wird eine Zeile länger als 80 Zeichen, so verschwindet ihr Anfang am linken Bildschirmrand. Es wird also nur diese eine Zeile horizontal "gescrollt".

Beim Verlassen einer Befehlszeile wird eine Syntaxkontrolle (s.o.) durchgeführt, die korrekte Einrückung des Befehls vorgenommen und die Zeile formatiert. Dazu gehört z.B. das Entfernen überflüssiger Leerzeichen (z.B. wird aus 2 + 2 dann 2+2) und das Schreiben von Befehlsworten und Variablennamen gemäß dem eingestellten DEFLIST. Das voreingestellte DEFLIST 0 führt z.B. dazu, daß alle Befehlsworte groß und alle Variablennamen klein geschrieben werden.

Der Cursor-Tasten-Block

Die Cursor-Steuerung kann über den Cursor-Tasten-Block erfolgen. Die Tastenbelegung ist:

```
Pfeil links  --> Cursor ein Zeichen nach links.
Pfeil rechts --> Cursor ein Zeichen nach rechts.
Pfeil oben   --> Cursor eine Zeile nach oben.
Pfeil unten  --> Cursor eine Zeile nach unten.
```

Die Bewegungen des Cursors unterliegen dabei einigen Einschränkungen. Er kann maximal ein Zeichen hinter das letzte Zeichen in einer Zeile gesetzt werden und maximal eine Zeile unter die letzte Programmzeile. Wenn der Cursor eine Zeile betritt, die kürzer ist als die aktuelle Cursor-Spaltenposition, so springt der Cursor an das Ende dieser Zeile, ansonsten behält er seine aktuelle Spaltenposition bei (im Unterschied zu den Editoren älterer GFA-BASIC-Versionen).

Es ist auch möglich, den Cursor mit der Maus zu positionieren. Zu diesem Zweck zeigt man mit dem Mauspfeil auf die gewünschte Position und drückt die linke Maustaste.

Das Betätigen der Insert-Taste fügt zwischen der Zeile über dem Cursor und der Cursor-Zeile eine Leerzeile ein, wenn in der aktuellen Zeile keine Änderung vorgenommen wurde. Der Cursor wird an den Anfang dieser Leerzeile gesetzt. Clr/Home bewegt den Cursor in die linke obere Ecke unterhalb der Menüleiste, Control + Clr/Home springt an den Anfang des Programm-Listings.

Mit Hilfe der Undo-Taste kann man Änderungen in einer Programmzeile wieder rückgängig machen, sofern die Zeile noch nicht verlassen wurde. Die Help-Taste ermöglicht es, Prozeduren im Programm-Listing "auf- und zuzuklappen".

Damit ist folgendes gemeint: Wenn man den Cursor auf eine Zeile positioniert, in der das Befehlswort PROCEDURE steht und dann die Help-Ta-

ste drückt, so werden alle Befehlszeilen bis zur nächsten RETURN-Anweisung (einschließlich) nicht mehr im Listing angezeigt. Statt dessen wird vor das Befehlswort PROCEDURE ein eckiger Pfeil und ein Leerzeichen gesetzt '> '.

In dieser Situation lassen sich auch keine Änderungen am Prozedurnamen oder der Parameterliste des Unterprogramms vornehmen. Um die Befehlszeilen zwischen PROCEDURE und RETURN wieder sichtbar zu machen, bewegt man den Cursor auf die mit dem eckigen Pfeil beginnende Zeile und drückt erneut die Help-Taste oder löscht das '>' Zeichen.

Dieses "Zusammenklappen" (Folding) eines Unterprogramms ermöglicht es, kurze übersichtliche Listings zu erstellen, in denen man nur das Unterprogramm "aufklappt", an dem man gerade arbeitet. Ein Programm mit zusammengeklappten Prozeduren kann z.B. so aussehen:

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE menueleiste
> PROCEDURE laden
> PROCEDURE speichern
> PROCEDURE bearbeiten
> PROCEDURE info_holen
> PROCEDURE info_zeigen
```

Mit einer aufgeklappten Prozedur sieht das Programm dann so aus:

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE menueleiste
```

```
PROCEDURE laden
  FILESELECT "\*.RSC","",rsc_datei$
  IF NOT EXIST(rsc_datei$)
    ALERT 1,"Datei existiert nicht !",1,"Abbruch",r%
  ELSE
    IF RSRC_LOAD(rsc_datei$)=0
      ALERT 1,"Fehler beim Laden der Datei !",1,"Abbruch",r%
      END
    ENDIF
  ENDIF
RETURN
> PROCEDURE speichern
> PROCEDURE bearbeiten
> PROCEDURE info_holen
> PROCEDURE info_zeigen
```

Der Zifferntastenblock

Der Zifferntastenblock dient normalerweise der Eingabe von Ziffern und einiger anderer Zeichen. Er kann aber auch in Verbindung mit der **Control-Taste** angesprochen werden. Seine Belegung entspricht dann weitgehend der Belegung bei Tastaturen mit eingeschaltetem NUMLOCK (z.B. bei PC's). Die Belegung ist:

Control und **4**	Cursor ein Zeichen nach links.
Control und **6**	Cursor ein Zeichen nach rechts.
Control und **8**	Cursor eine Zeile nach oben.
Control und **2**	Cursor eine Zeile nach unten.
Control und **7**	Springe an den Programmanfang.
Control und **1**	Springe an das Programmende.
Control und **9**	Eine Seite nach oben "blättern".
Control und **3**	Eine Seite nach unten "blättern".
Control und **0**	Entspricht Insert.
Control und **.**	Entspricht Delete.

Der Zifferntastenblock kann auch in einen Modus geschaltet werden, in dem diese Tasten auch ohne Control wirksam sind. Die Umschaltung erfolgt über Control (-Kombinationen) anderer Tasten des Ziffernblocks.

Control und **-** sowie **Control** und **(** entsprechen NUMLOCK. Danach ergibt das Drücken einer Taste des Zifferntastenblocks das oben mit Control aufgeführte Kommando. Der NUMLOCK-Modus kann durch erneutes Drücken derselben Tasten wieder aufgehoben werden.

Weitere Editiertasten auf dem Haupttastaturfeld

Mit Hilfe der **Delete-Taste** kann das Zeichen, auf dem der Cursor steht, gelöscht werden; der Rest der Zeile wird nachgezogen. Die **Backspace-Taste** löscht das Zeichen, das links vom Cursor steht und zieht den Rest der Zeile nach.

Das Drücken der **Tab-Taste (Tabulator)** läßt den Cursor an die nächste rechte Tabulatorposition springen. Diese Positionen haben acht Zeichen Abstand. **Control + Tab** bewegt den Cursor eine Tabulatorposition nach links.

Die **Return-** und die **Enter-Taste** lassen den Cursor an den Anfang der nächsten Zeile springen. Durch das Drücken der **Escape-Taste** gelangt man in den Direktmodus.

Control-Kommandos

In den anderen Abschnitten wurden viele Control-Kommandos bereits genannt. Hier sollen diese Kommandos und die dort nicht genannten **Control-Tasten-Befehle** noch einmal zusammengefaßt werden (mit Ausnahme der Control-Kommandos, die mit dem Zifferntastenblock zusammenarbeiten):

```
Control+Delete  Löscht die Zeile, in der der Cursor steht.
Control+U       (undelete)
```

fügt diejenige Zeile ein, die als letzte mit **Control + Delete** oder **Control + Y** gelöscht wurde. Dies dient zum einen der Restaurierung von Zeilen, die versehentlich gelöscht wurden. Zum anderen können auf diese Art und Weise sehr leicht einzelne Zeilen verschoben oder kopiert werden (z.B. kopieren: **Control-Taste** festhalten und dann **Delete** und **zweimal U** drücken).

Control+Y	Löscht die Zeile, in der der Cursor steht.
Control+N	Fügt eine Leerzeile oberhalb der Befehlszeile ein, in der der Cursor steht (wie Insert), auch dann, wenn in der Zeile Änderungen vorgenommen wurden.
Control+Q	Blockmenü aufrufen (wie Funktionstaste F4).
Control+B	Blockanfang markieren (Block).
Control+K	Blockende markieren (BlocK).
Control+R	"Blättert" eine Seite nach oben.
Control+C	"Blättert" eine Seite nach unten.
Control+E	Text ersetzen.
Shift+Control+E	Ersetzen mit Abfrage von Such- + Ersatz-String.
Control+F	(find), Text suchen.
Shift+Control+F	Text suchen mit Abfrage des Such-Strings.
Control+Cursor links	Springt an den Anfang der Zeile.
Control+Cursor rechts	Springt an das Ende der Zeile.
Control+Cursor oben	"Blättert" eine Seite nach oben.
Control+Cursor unten	"Blättert" eine Seite nach unten.
Control+Clr/Home	Springt an den Anfang des Programms.
Control+Z	Springt an das Ende des Programm-Listings.
Control+Tab	Springt eine Tabulatorposition nach links.
Control+G (goto)	Öffnet das Zeilennummernanzeigefeld für die Eingabe einer Zeilennummer (die entsprechende Zeile wird dann angesprungen).

Eine besondere Gruppe von **Control-Kommandos** ermöglicht das Setzen und Anspringen von Marken im Editor. Diese Marken gelten nur für den Editor, sie haben also nichts mit den Marken zu tun, die von GOTO oder RESTORE benutzt werden. Diese Editor-Marken können an der Cursor-Position durch Drücken von **Control** und einer **Ziffer der Haupttastatur** gesetzt werden. Anschließend kann durch das Drücken von **Alternate** und dieser **Ziffer** an die gewünschte Marke gesprungen werden.

Dabei sind die Tastenkombinationen von **Alternate** mit den Ziffern **7-9** und **0** vorbelegt. Das Drücken von **Alternate** und **7** bewirkt das Springen zur letzten Cursor-Position vor dem Umschalten in den Direkt-Modus oder vor dem letzten Programmstart. **Alternate** und **8** springt an die Stelle, an der der Cursor beim Start des Editors war. Bei **Alternate** und **0** wird zur letzten Cursor-Position gesprungen, an der eine Änderung vorgenommen wurde. **Alternate** und **9** springt an die Stelle, an der der letzte Suchvorgang gestartet wurde.

Die Menüleiste und die Funktionstasten

In den obersten beiden Bildschirmzeilen des Editors befinden sich zwei Menüzeilen. Ganz links steht das ATARI-Symbol, bei dessen Anklicken ein Pull-Down-Menü herunterklappt. Dieses enthält die Menütitel ATARI-Symbol und GFA-BASIC.

Unter dem ATARI-Symbol erscheint beim Anklicken des Menüpunktes GFA-BASIC 3.00 D die Versionsnummer mit den zwei Auswahlmöglichkeiten Editor und Menu. Klickt man den Editor-Knopf an, gelangt man zurück in den GFA-BASIC-Editor. Bei der Wahl von Menu kommt man zurück in das Pull-Down-Menü.

Der Menütitel GFA-BASIC enthält die folgenden Menüpunkte:

Save — Springt in den Editor, danach erscheint eine File-Select-Box, und es kann ein Dateiname angegeben werden, unter dem das aktuelle Programm abgespeichert werden kann.

Load — Springt in den Editor, woraufhin in der erscheinenden File-Select-Box der Name einer Datei angegeben werden kann, die geladen werden soll.

Deflist — Ermöglicht das Einstellen des Deflist-Modus über eine Alert-Box (vergleiche Abschnitt Spezielles).

Neue Namen — Mit Hilfe dieses Menüpunktes kann in einen Modus geschaltet werden, in dem die Einführung neuer Variablen in das aktuelle Programm bestätigt werden muß. Dieser Modus ist in der Voreinstellung ausgeschaltet. Ist er aktiv, so erscheint bei der Verwendung einer noch nicht im Programm eingeführten Variablen oder auch eines Prozedur- bzw. Funktionsnamens eine Alertbox, in der man angeben kann, daß man diese neue Variable in das Programm übernehmen will oder daß es sich um einen Schreibfehler handelt (in diesem Fall erscheint die Fehler-Meldung "Syntax Error").

Am rechten Rand der Menüleiste befindet sich oben eine Uhr und darunter die Anzeige der aktuellen Textzeilennummer. Der Umgang mit diesen beiden Menüleistenelementen wird am Ende dieses Abschnitts erklärt.

Zwischen dem ATARI-Symbol und der Uhr stehen zwanzig Kommandoworte, jeweils zwei übereinander, die entweder mit der Maus angeklickt oder über die **Funktionstasten** angesprochen werden können. Die untere Zeile der Kommandoworte läßt sich durch das Drücken einer **Funktionstaste** ansprechen; die Kommandos der oberen Zeile durch das Drücken einer **Shift-Taste** und der entsprechenden **Funktionstaste**. Zu dem Kommandowort unten links (Load) gehört z.B. die **F1-Taste**, das Kommandowort darüber (Save) läßt sich mit SHIFT + F1 aufrufen.

Unter dem ATARI-Symbol ist Platz für zwei weitere Zeichen. Dort wird der Status der **CAPSLOCK-** und **NUMLOCK-Taste** angezeigt. Ist die **CAPSLOCK-Taste** gedrückt, erscheint links unter dem ATARI-Symbol ein nach oben gerichteter Pfeil. Bei gedrückter **NUMLOCK-Taste**, d.h. **Control** + (, steht rechts unter dem ATARI-Symbol das Exponent-Zeichen ^. NUMLOCK- und CAPS-LOCK-Modus können auch durch Klicken mit der linken **Maustaste** unter dem ATARI-Symbol angewählt werden.

Die Menüpunkte haben folgende Bedeutung:

Load **F1**

Das Kommando Load ermöglicht das Laden eines GFA-BASIC-3.0 Programms. Das hier verwendete Format benutzt sogenannte Befehlstokens. Eine solche Programmdatei kann besonders schnell geladen und gespeichert werden. Für dieses Format wird normalerweise die Dateiendung (Extension) .GFA benutzt.

Die älteren GFA-BASIC-Versionen benutzen ein anderes Token-Format. Ihre Programme müssen deshalb in der alten Version mit Save,A abgespeichert und in GFA-BASIC 3.0 mit Merge geladen werden. Danach können sie in 3.0 mit Save abgespeichert und mit Load geladen werden.

Save **Shift + F1**

Es erscheint eine File-Select-Box, in der ein Name angegeben werden kann. Unter diesem Namen wird dann das Programm gespeichert, das sich gerade im Editor befindet. Es wird dabei das unter Load genannte tokenisierte Format benutzt. Die voreingestellte Dateikennung (Extension) heißt .GFA. Sie wird, wenn der Benutzer keine Extension angibt, automatisch an den Dateinamen angehängt.

Besteht bereits eine Datei unter dem angegebenen Namen, wird diese unter dem gleichen Dateinamen mit der Extension .BAK abgespeichert.

Merge **F2**

Mit diesem Kommando kann eine Datei im ASCII-Format in das aktuelle Programm eingefügt werden. Die Einfügung erfolgt dabei vor der Zeile, in der der Cursor steht. Die voreingestellte Dateikennung ist .LST. Programme aus älteren GFA-BASIC-Versionen müssen in der alten Version mit Save,A abgespeichert und in GFA-BASIC 3.0 mit Merge neu geladen werden.

= = > Programmzeilen mit dieser Kennung werden beim Mergen eines
= = > Programms erzeugt, das der Interpreter nicht versteht.

Save,A **Shift + F2**

Mit Save,A kann das aktuelle Programm im ASCII-Format abgespeichert werden. Eine in diesem Format gespeicherte Datei kann mit Merge wieder geladen werden. Die voreingestellte Dateiendung (Extension) ist .LST. Sie wird, wenn der Benutzer keine Extension angibt, automatisch an den Dateinamen angehängt.

Besteht bereits eine Datei unter dem angegebenen Namen, wird diese unter dem gleichen Dateinamen mit der Extension .BAK abgespeichert.

Llist F3

Dieser Befehl leitet den Ausdruck des Programms ein, das sich gerade im Editor befindet. Die Formatierung des Ausdrucks kann durch die sogenannten Punktbefehle beeinflußt werden. Diese Befehle werden wie normale Befehlszeilen in das Programm-Listing eingefügt. Sie lauten (x ist ein Platzhalter für eine Ziffer):

```
.ll xx        - Maximale Zeilenlänge
.pl xx        - Maximale Seitenlänge
              - Seitenvorschub ausführen
              - conditional page
              - Zeilennumerierung
              - Linker Rand
.ff xxx       - Form-Feed-Zeichenkette (für Drucker, die andere Werte
                für Form-Feed haben, kann ein Ersatz-String angegeben
                werden. Voreingestellt ist .ff\012)
.he kopf      - Text der Kopfzeile
.fo fuß       - Text der Fußzeile
.lr xx        - Linker Rand
.l-           - Diese Anweisung bewirkt, daß die nachfolgenden Zeilen
                nicht mit aufgelistet werden. Das Listing kann durch ein
                später im Programm folgendes .l+ wieder eingeschaltet
                werden.
.l+             siehe .l-
.n1 bis .n9     Schaltet Zeilennumerierung mit ein- bis neun Stellen ein.
.n0             Schaltet die Zeilennumerierung aus.
```

In den Text der Kopf- und Fußzeile sowie den FF-String können noch besondere Platzhalter eingefügt werden.

```
\xxx          - Zeichen mit dem ASCII-Code xxx
\d            - Datum
\t            - Uhrzeit
#             - Seitennummer
```

Um die Sonderzeichen # und \ ausgeben zu können, muß man ein weiteres \-Zeichen davor setzen. Die Zeichenkombination \\ gibt also ein \ auf dem Drucker aus, \# ein #.

Quit **Shift + F3**

Ermöglicht das Verlassen des GFA-BASIC-Interpreters.

Block **F4**

Wenn kein Block markiert ist und dieser Menüpunkt gewählt wird, so erscheint in der Menüleiste der Text "Block ???", um anzuzeigen, daß die Wahl des Block-Kommandos in dieser Situation unsinnig ist. Ist aber ein Block markiert, so wird in die oberste Bildschirmzeile ein Menü eingeblendet. Die einzelnen Punkte dieses Menüs können mit der Maus angeklickt werden, sie können aber auch durch das Drücken eines **Kommandobuchstabens** aktiviert werden. Die Menüpunkte sind (in Klammern steht jeweils die Taste, mit der der Menüpunkt aufgerufen werden kann):

Copy	Kopiert den Block an die aktuelle Cursor-Position. Der Block bleibt dabei markiert.
Move	Verschiebt den Block an die aktuelle Cursor-Position. Die Blockmarkierung wird dabei entfernt.
Write	Speichert den Block als ASCII-Datei ab.
Llist	Druckt den Block aus.
Start	Springt an den Anfang des Blocks.
End	Springt an das Ende des Blocks.
^Del	Löscht den Block (Control-D).
Hide:	Entfernt die Blockmarkierung.

Das Drücken einer anderen Taste (oder einer Maustaste) außerhalb dieses Bereiches entfernt das Blockmenü.

New **Shift + F4**

Löscht das Programm, das sich gerade im Editor befindet.

BlkEnd **F5**

Mit BlkEnd kann die Zeile, die vor der Cursor-Zeile steht, als Blockende markiert werden. Befindet sich die Blockanfangmarkierung über dieser Zeile, so wird der Block in der Monochromversion mit einem Punktraster unterlegt. In der Farbversion wird der markierte Block in einer anderen Farbe dargestellt. Dieses Kommando kann auch durch Drücken von **Control + K** ausgelöst werden.

BlkSta **Shift + F5**

Mit BlkSta wird die Zeile, in der der Cursor steht, als Blockanfang markiert. Ist hinter dieser Zeile bereits eine Blockendemarkierung gesetzt worden, so wird der Blockbereich mit einem Punktraster unterlegt. Dieses Kommando kann auch durch das Drücken von **Control + B** ausgelöst werden.

Find **F6**

Nach dem Auslösen des Find-Kommandos kann man einen Text angeben, nach dem gesucht werden soll. Die Suche beginnt in der Zeile, in der der Cursor steht. Das Suchen kann mit **Control + F** oder mit **Control + L** erneut aufgerufen werden, ohne daß die Abfrage des zu suchenden Textes erscheint.

Wurde der zu suchende Text gefunden, so steht der Cursor am Anfang der Zeile, in der der Suchtext enthalten ist; wurde er nicht gefunden, dann steht der Cursor am Programmende.

Wenn die Find-Anweisung erneut aufgerufen wird, so steht in der Zeile, in der der Such-String eingebenen werden kann, der beim letzten Mal benutzte Such-String. Bei der Eingabe des Suchtextes stehen folgende Editiermöglichkeiten zur Verfügung:

Cursor links Cursor geht ein Zeichen nach links (sofern er nicht ganz links steht).

- rechts Cursor geht ein Zeichen nach rechts (sofern er nicht am Ende des Such-Strings steht).

Delete Löscht das Zeichen des Such-Strings, das unter dem Cursor steht, der Rest des Such-Strings wird von rechts nach links nachgerückt.

Backspace Löscht das Zeichen links vom Cursor (sofern dieser nicht schon ganz links steht) und zieht den Rest des Such-Strings nach.

Escape Löscht das Eingabefeld.

Return/Enter Bestätigt den eingebenen Suchtext und beginnt mit der Suche.

Das Find-Kommando kann auch durch die Tastenkombination **Shift + Control + F** oder **Shift + Control + L** aufgerufen werden. In "zusammengeklappten" (siehe **Help-Taste** im Abschnitt Cursor-Tasten-Block) Prozeduren wird kein Text gefunden.

Replac **Shift + F6**

Dieses Kommando dient dem Ersetzen eines Textes durch einen anderen. Nach seinem Aufruf wird zunächst nach dem Text gefragt, der ersetzt werden soll. Anschließend wird nach dem Text gefragt, mit dem zu ersetzen ist. Nun wird im Programm ab der Cursor-Position mit der Suche nach dem zu ersetzenden Text begonnen. Wenn der Suchtext gefunden wird, so springt der Cursor an den Anfang der entsprechenden Zeile.

Nun kann die Ersetzung durch das Drücken der Tastenkombination **Control + E** vorgenommen werden. Das weitere Drücken dieser Tastenkombination führt dazu, daß die nächste ersetzbare Textstelle aufgesucht wird oder, falls sie sich in der aktuellen Cursor-Zeile befindet, ersetzt wird.

Das Replac-Kommando kann auch durch die Tastenkombination **Shift + Control + E** aufgerufen werden. Die Editiermöglichkeiten, die bei der Eingabe des Such- und des Ersatztextes zur Verfügung stehen, wurden bei der Beschreibung des Find-Befehls aufgeführt. In "zusammengeklappten" (siehe **Help-Taste** im Abschnitt Cursor-Tasten-Block) Prozeduren wird kein Text gefunden.

Pg Down F7

"Blättert" den Programmtext um eine Bildschirmseite nach unten. Kann auch durch **Control + C** aufgerufen werden.

Pg Up Shift + F7

"Blättert" den Programmtext um eine Bildschirmseite nach oben. Kann auch durch **Control + R** aufgerufen werden.

Insert/Overwr F8

Schaltet zwischen dem Einfüge- und dem Überschreibmodus hin und her.

Txt 16/Text 8 Shift + F8

Dieses Kommando steht nur in der Monochromauflösung zur Verfügung. Im Modus 16 werden Buchstaben mit einer Höhe von 16 Pixeln dargestellt, so daß gleichzeitig 23 Zeilen auf dem Bildschirm sichtbar sind. Im Modus 8 ist die Buchstabenhöhe nur 8 Pixel, aber es können 48 Zeilen gleichzeitig angezeigt werden. Mit Txt 16/Text 8 kann zwischen diesen beiden Modi hin- und hergeschaltet werden.

Flip **F9**

Mit diesem Kommando kann auf den Ausgabebildschirm umgeschaltet werden, auf dem die Situation zu sehen ist, die das letzte aufgerufene Programm hinterließ. Durch das Drücken einer **Taste** (auch **Maustaste**) kann wieder in den Editor-Bildschirm zurückgeschaltet werden.

Direct **Shift + F9**

Dieses Kommando ruft den Direktmodus auf. In diesem Modus können GFA-BASIC-Befehle eingegeben werden, die dann nach dem Drücken der Return- oder Enter-Taste sofort abgearbeitet werden. Einige Befehle, z.B. Schleifenbefehle, sind im Direktmodus allerdings nicht aufrufbar. Diesen Modus kann man auch durch das Drücken der Escape-Taste aufrufen.

Außerdem ist es im Direktmodus möglich, mit den Tasten Cursor hoch und Cursor runter die letzten acht Befehle (die im Direktmodus eingegeben wurden) wiederzuholen.

Das Drücken der Undo-Taste bewirkt die Anzeige des jeweils letzten Befehls. Mit den Tastenkombinationen Control + Cursor links bzw. Control + Cursor rechts lassen sich der Zeilenanfang und das Zeilenende erreichen. Zusätzlich ist es durch Drücken der Insert-Taste möglich, zwischen Einfüge- und Überschreibmodus hin- und herzuschalten.

Durch das Drücken der Escape-Taste, gefolgt von Return, kann man de Direktmodus wieder verlassen. Alternativ dazu ist das Verlassen auch durch das gleichzeitige Drücken von **Control + Shift + Alternate** möglich.

Um mehrzeilige Anweisungen aufzurufen, kann man im Editor eine Prozedur schreiben, die diese Befehle beinhaltet. Im Direktmodus kann diese Prozedur dann aufgerufen werden.

Test **F10**

Nach der Auslösung dieser Anweisung wird geprüft, ob alle Schleifen, Unterprogramme und bedingte Anweisungen des aktuellen Programms geschlossen sind. Es wird also eine Strukturprüfung vorgenommen.

Mit Hilfe von Control, linker Shift-Taste und Alternate kann ein laufendes Programm unterbrochen werden.

Run **Shift + F10**

Startet das Programm, das sich gerade im Editor befindet. Enthält dieses Programm einen strukturellen Fehler, z.B. eine nicht abgeschlossene FOR-NEXT-Schleife, so wird eine entsprechende Fehlermeldung ausgegeben und das Programm nicht gestartet.

Control + Shift + Alternate

Mit Hilfe der Tastenkombination Control + linke Shift-Taste + Alternate kann das laufende Programm unterbrochen werden.

Die Zeilenanzeige und die Uhr

Rechts unten in der Menüleiste befindet sich eine Anzeige, in der die Nummer der Zeile steht, in der sich der Cursor gerade befindet. In diese Zeilenanzeige kann auch eine Zeilennummer eingetragen werden, so daß der Cursor in die entsprechende Zeile springt. Zu diesem Zweck kann das Anzeigefeld entweder mit der Maus angeklickt oder durch das Kommando **Control + G** geöffnet werden.

Es werden in diesem Feld nur Ziffern als Eingabe akzeptiert. Die Editiermöglichkeiten für den Eintrag der gewünschten Zeilennummern sind:

Cursor links	Cursor geht ein Zeichen nach links (sofern er nicht ganz links steht).
- rechts	Cursor geht ein Zeichen nach rechts (sofern er nicht am Ende des Eingabefeldes steht).
Backspace	Hat dieselbe Funktion wie Cursor links.
Escape	Löscht das Eingabefeld.
Return/Enter	Bestätigt die eingegebene Zahl.

Über der Zeilennummernanzeige ist eine Uhr, die die Systemzeit anzeigt. Nach dem Anklicken der Uhr mit dem Mauspfeil hat man die Möglichkeit, die Uhr neu zu stellen. Die Editiermöglichkeiten für die Eingabe der Uhrzeit sind dieselben wie bei der Eingabe ins Zeilennummernfeld. Lediglich die **Escape-Taste** hat hier eine andere Bedeutung, da sie die Eingabe abbricht, wobei der alte Wert für die Uhrzeit erhalten bleibt.

Spezielles

DEFBIT f$
DEFBYT f$
DEFINT f$
DEFWRD f$
DEFFLT f$
DEFSTR f$

f$: String-Konstante

Die Anweisung DEFxxx dient der vereinfachten Variablendeklaration. Dabei ist xxx ein Platzhalter für die Abkürzungen von Variablentypen, die in der folgenden Tabelle aufgeführt und erläutert werden:

DEFxxx	f$	Postfix	Alle Variablen...
DEFBIT	"b"	!	mit dem Anfangsbuchstaben 'b' werden als Bool'sche Variablen deklariert.
DEFBYT	"by"	|	mit den beiden Anfangsbuchstaben 'by' werden als 1-Byte-Integer deklariert.
DEFWRD	"w"	&	mit dem Anfangsbuchstaben 'w' werden als 2-Byte-Integer mit Vorzeichen deklariert.
DEFINT	"i-k,m-p"	%	mit den Anfangsbuchstaben 'i' bis 'k' und 'm' bis 'p' werden als 4-Byte-Integer mit Vorzeichen deklariert.
DEFFLT	"x-z"	#	mit den Anfangsbuchstaben 'x' bis 'z' werden als 8-Byte-Fließkommawert deklariert.
DEFSTR	"s,t"	$	mit dem Anfangsbuchstaben 's' und 't' werden als Zeichenkette deklariert.

Die Angabe von DEFSNG oder DEFDBL wird vom Editor automatisch durch DEFFLT ersetzt.

Es ist normalerweise sinnvoll, solche globalen Deklarationen zu Programmbeginn durchzuführen. Soll an irgendeiner Stelle des Programms von der mit DEFxxx festgelegten Definition abgewichen werden, muß nur das gewünschte Postfix hinter der Variable angegeben werden. Diese explizite Angabe des Variablentyps hat immer Vorrang vor den globalen Definitionen.

Die Deklaration von Variablentypen ist solange gültig, bis eine andere Deklaration vorgenommen wird. Der voreingestellte Variablentyp ist ein 8-Byte-Fließkommawert.

Die Anzeige der Variablentypen sowie die Groß- und Kleinschreibung im Listing regelt der DEFLIST-Befehl.

DEFLIST n

n: iexp

DEFLIST legt das Format des Programm-Listings fest. Der numerische Ausdruck n kann einen Wert zwischen 0 und 3 (einschließlich) annehmen. Die Wirkung des DEFLIST-Befehls auf die Schreibweise der Befehls- und Variablennamen wird in der folgenden Tabelle dargestellt:

n	Befehl	Variable
0	PRINT	abc
1	Print	Abc
2	PRINT	abc#
3	Print	Abc#

Der voreingestellte Modus ist DEFLIST 0.

DEFLIST 0

GFA-BASIC-Befehle und -Funktionen werden in Großbuchstaben dargestellt. Variablen-, Prozedur- und Funktionsnamen werden klein geschrieben.

DEFLIST 1

Sowohl GFA-BASIC-Befehle und Funktionen als auch Variablen-, Prozedur- und Funktionsnamen werden mit großen Anfangsbuchstaben dargestellt, und der Rest des Namens wird klein geschrieben.

DEFLIST 2

Wie DEFLIST 0, aber zusätzlich wird bei **allen** Variablennamen ein Kennzeichen (Postfix) angehängt.

DEFLIST 3

Wie DEFLIST 1, aber zusätzlich wird bei **allen** Variablennamen ein Kennzeichen (Postfix) angehängt.

$ text

```
text: Folge von beliebigen Zeichen
```

Der Befehl $, der vom Interpreter wie ein REM behandelt wird, dient zur Steuerung des Compilers. Eine genaue Beschreibung finden Sie in der Anleitung der Compiler-Version zum GFA-BASIC 3.0.

2 - Variablen und Speicherverwaltung

Variablentypen

GFA-BASIC 3.0 verfügt über folgende Variablentypen:

Name	Postfix	Speicherplatzbedarf
Boolean	!	1 Byte (in Arrays ein Bit)
Byte	|	1 Byte
Word	&	2 Byte
Integer	%	4 Byte
Float	#	8 Byte
String	$	(Abhängig von der String-Länge)

Bool'sche Variablen können nur die Werte 0 (FALSE) oder -1 (TRUE) annehmen. Wird ihnen ein Wert zugewiesen, der ungleich Null ist, so wird dieser Wert als -1 aufgefaßt. Dieser Variablentyp wird mit dem Postfix ! versehen und belegt 1 Byte Speicherplatz. In bool'schen Arrays benötigt ein Arrayelement lediglich ein Bit.

```
Beispiele: b!=TRUE
           c!=x>y
```

Der Variablentyp Byte kann Werte zwischen 0 und 255 annehmen. Größere Werte für x werden als Überlauf behandelt. Das Postfix dieses Variablentyps ist der senkrechte Strich |. Wie der Name schon sagt, belegt dieser Variablentyp ein Byte.

```
Beispiel: x|=128
```

Word ist ein vorzeichenbehafteter 2-Byte-Integertyp. Das Postfix dieses Typs ist &.

Der darstellbare Zahlenbereich reicht von -32768 bis 32767.

```
Beispiel: x&=32767
```

Integer ist ein vorzeichenbehafteter 4-Byte-Integertyp. Als Postfix wird das Zeichen % benutzt.

Der darstellbare Zahlenbereich reicht von -2147483648 bis 2147483647.

```
Beispiel: x%=2000000000
```

Float ist ein Fließkommavariablentyp mit 8 Byte Speicherplatzbedarf. Bei diesem Typ wird kein Postfix verwendet oder das #-Zeichen. Der darstellbare Zahlenbereich reicht von 2.225073858507E -308 bis 3.595386269725E +308.

Zeichenketten (Strings) werden mit dem Postfix $ gekennzeichnet. Sie können eine maximale Länge von 32767 Zeichen haben. Strings werden mit Hilfe eines sogenannten Deskriptors verwaltet. Ein solcher Deskriptor hat eine Länge von sechs Byte. Die ersten vier Bytes enthalten die Adresse der Zeichenkette, die letzten beiden Bytes die Länge des Strings. Hinter der Zeichenkette wird ein Füllbyte eingefügt, falls die Länge des Strings ungerade ist. Danach folgt die Adresse des Deskriptors (Backtrailer).

Die Adressen aller Variablen-Typen können mit Hilfe der Funktionen VARPTR (oder V:) und ARRPTR (oder *) ermittelt werden. Bei Strings erhält man mit VARPTR die Adresse des ersten Bytes der Zeichenkette, mit ARRPTR erhält man die Adresse des String-Deskriptors.

Bei Arrays kann man mit VARPTR / V: die Adressen der einzelnen Array-Elemente ermitteln (z.B. V: x%(5)). Mit Hilfe von ARRPTR / * erhält man die Adresse der Array-Deskrptors (z.B. ARRPTR (x%())). Mit VARPTR / V: kann man also die Adresse einzelner Feldelemente ermitteln, während ARRPTR die Adresse des Feld-Deskriptors bestimmt.

Felder

DIM, DIM?
OPTION BASE
ARRAYFILL

Von allen Variablentypen lassen sich Felder erzeugen. Das Einrichten eines solchen Feldes erfolgt mit Hilfe des DIM-Befehls. Die Nachfragefunktion DIM? gibt an, wie viele Elemente ein Array enthält.

Die Verwaltung von Arrays im Speicher erfolgt über Deskriptoren. Ein solcher Deskriptor ist eine sechs Byte lange Struktur, in dessen ersten vier Bytes die Adresse des Feldes enthalten ist. Die nächsten beiden Bytes geben an, wie viele Dimensionen das Feld hat. Das Feld beginnt mit jeweils vier Byte für jede Arraydimension, wobei mit der letzten Dimension begonnen wird. Dahinter folgen dann die Inhalte der Feldelemente. Bei String-Variablen sind dies die Deskriptoren der Zeichenketten.

Beispielsweise ergibt sich nach DIM a%(2,3) folgende Situation; Mit Hilfe von *a%() erhält man die Adresse des Felddeskriptors. Die Anzahl der Dimensionen des Arrays steht in den letzten beiden der sechs Deskriptor-Bytes, also ergibt INT{*a%()+4} den Wert 2. Das Array beginnt mit der Anzahl der Unterteilungen der zweiten Dimension, wobei das nullte Element mitgezählt wird (vorausgesetzt, OPTION BASE 0 gilt). Dahinter steht die Anzahl der Unterteilungen der ersten Dimension. Dann ergibt

```
{{*a%()}}     den Wert 4 und
{{*a%()}+4}   den Wert 3.
```

Daran schließen sich die 12 Array-Elemente an, mit jeweils 4 Byte Speicherplatzbedarf in der Reihenfolge:

```
a%(0,0)  a%(1,0)  a%(2,0)  a%(0,1)  a%(1,1)  a%(2,1)  usw.
```

DIM x(d1, [d2,...]) [,y(d1, [d2,...])]
DIM?(x())

```
x,y: Variablenname (beliebiger Variablentyp)
d1,d2: iexp
```

Mit Hilfe des Befehls DIM können numerische und Zeichenkettenfelder eingerichtet werden. Der Aufbau eines solchen Arrays wurde in der Einführung dieses Abschnitts beschrieben.

Aus praktischer Sicht ist die mögliche Anzahl der Dimensionen des Arrays nicht begrenzt. Die Anzahl der Array-Elemente bei mehrdimensionalen Arrays ist insofern beschränkt, als daß die letzte Dimension kleiner als 65535 sein muß. Ebenso muß das Produkt aus der Anzahl der Feldelemente kleiner als 65535 sein. (So ist z.B. DIM a%(100,10,10) erlaubt, da die letzte Dimension (10) und das Produkt aus der Anzahl der Feldelemente (100*10*10 = 10000) kleiner als 65535 sind.)

Mit Hilfe von OPTION BASE kann festgelegt werden, ob es ein nulltes Array-Element gibt. In einem Array können nur Variablen desselben Typs existieren. Das Ansprechen der Arrayelemente erfolgt über deren Indizes. Die Funktion DIM? ermittelt, wie viele Elemente das Array hat.

Beispiel:

```
DIM x(10)
x(4)=3
PRINT x(LEN("Test"))
PRINT DIM?(x())
DIM y%(2,3)
PRINT DIM?(y%())
```

--> Richtet zwei Arrays ein. Schreibt in eines der Arrays einen Wert und liest ihn wieder aus. Schreibt die Anzahl der Elemente der beiden Arrays auf den Bildschirm (11 und 12).

OPTION BASE 0
OPTION BASE 1

Mit Hilfe des Befehls OPTION BASE kann bestimmt werden, ob bei Arrays ein nulltes Element existieren soll oder nicht. Mit OPTION BASE 0 kann ein solches nulltes Element eingerichtet werden, mit OPTION BASE 1 wird es verhindert. In diesem Fall beginnen die Array-Indizes bei 1.

Der Inhalt von Feldern wird durch den OPTION BASE-Befehl nicht verändert, allerdings ändert sich unter Umständen der Index der einzelnen Elemente um den Wert 1.

Wird kein OPTION BASE-Befehl gegeben, so gibt es Array-Elemente mit dem Index 0.

Beispiel:

```
DIM x%(3)
FOR i%=3 DOWNTO 0
  x%(i%)=i%
  PRINT i%,x%(i%)
NEXT i%
OPTION BASE 1
FOR i%=3 DOWNTO 0
  PRINT i%,x%(i%)
NEXT i%
```

--> Schreibt die Indizes und den Inhalt des Arrays x%() auf den Bildschirm. Das Programm endet mit der Fehlermeldung 'Feld-Index falsch', da x%() nach OPTION BASE 1 kein nulltes Element mehr besitzt.

ARRAYFILL x(),y

```
x: Name eines Arrays mit numerischem Variablentyp
y: aexp
```

Der Befehl ARRAYFILL setzt alle Elemente des Arrays x gleich dem Wert des numerischen Ausdrucks y.

Beispiel:

```
DIM x(10)
PRINT x(4)
ARRAYFILL x(),5+1
PRINT x(4)
```

--> Gibt die Zahl Null aus, da beim DIM-Befehl automatisch alle Feldelemente auf Null gesetzt werden. Nach Füllen des Arrays wird die Zahl 6 auf den Bildschirm ausgegeben.

Typumwandlung

TYPE(x)

x: iexp

Mit Hilfe der Funktion TYPE kann überprüft werden, ob ein korrekter Pointer auf eine Variable übergeben wurde. Es wird eine Kennzahl für den Typ dieser Variable oder der Wert -1 zurückgegeben. Man erhält, je nach Variablentyp, auf den x zeigt, folgende Werte:

```
Float          -->  0
String         -->  1
Integer        -->  2
Boolean        -->  3
Float-Array    -->  4
String-Array   -->  5
Integer-Array  -->  6
Bool'sches Array -->  7
Word           -->  8
Byte           -->  9
Word-Array     --> 12
Byte-Array     --> 13
```

Beispiel:

```
a$="test"
x%=4
DIM y(3)
PRINT TYPE(*a$),TYPE(*y())
```

--> Schreibt die Zahlen 1 und 4 auf den Bildschirm.

ASC(a$)
CHR$(x)

```
a$: sexp
x: aexp
```

ASC und CHR$ sind Umkehrfunktionen zueinander.

Die Funktion ASC liefert den ASCII-Code des ersten Zeichens im String a$. Ist a$ ein Leer-String, so wird eine Null zurückgemeldet.

CHR$ liefert das ASCII-Zeichen, das den Code x hat. Dabei wird nur das Lowbyte von x bewertet (entspricht dem Ausdruck x AND 255).

Beispiel:

```
PRINT ASC("TEST")
code|=ASC(CHR$(65))  ! CHR$(65) ergibt 'A'
PRINT code|,CHR$(189)
```

--> Schreibt die Zahlen 84, 65 und das Copyright-Zeichen auf den Bildschirm.

Die Funktionen STR$, BIN$, OCT$ und HEX$ wandeln einen numerischen Ausdruck x in eine Zeichenkette um. In dieser Zeichenkette steht die Zahl, die aus dem numerischen Ausdruck x berechnet wird.

In y steht, wie lang der zurückgemeldete String sein soll. Ist y größer als die Anzahl der Zeichen, die für die Darstellung der Zahl benötigt werden, so wird der String bei STR$ vorn mit Leerzeichen aufgefüllt. Bei den Funktionen BIN$, OCT$ und HEX$ wird der Ergebnis-String gegebenenfalls mit Nullen aufgefüllt.

STR$(x)
STR$(x,y)
STR$(x,y,z)

```
x,y,z: aexp
```

STR$ wandelt eine Zahl 'x' in einen String. Ein zweiter Parameter 'y' gibt an, auf wie viele Stellen der Ergebnis-String links mit Leerzeichen aufgefüllt wird bzw. wie viele Zeichen des Ergebnis-Strings (vom Zeichenkettenende gezählt) verwendet werden.

STR$(x,n) entspricht RIGHT$(SPACE$(n) + STR$(x),n).

Eine weitere Variante von STR$ verfügt über einen dritten Parameter 'z'. Dabei wird die Zahl 'x' auf 'y' Stellen mit 'z' Nachkommastellen formatiert und gerundet.

Beispiel:

```
a=123.4567
PRINT STR$(a,6,2)
PRINT STR$(PI,5,3)
PRINT STR$(PI,2,2)
```

--> Schreibt die Zahlen 123.46, 3.142 und 14 auf den Bildschirm.

BIN$(x [,y])
OCT$(x [,y])
HEX$(x,y)

```
x,y: iexp
```

BIN$ wandelt eine Integerzahl in die binäre Darstellung.

Das bedeutet, daß die Zahl nach der Umwandlung im Zahlensystem zur Basis 2 vorliegt. Dabei werden Zahlen nur durch die Ziffern 0 und 1 dargestellt. Der Parameter 'y' gibt an, wie viele Stellen (1 bis 32) verwendet werden.

OCT$ wandelt einen Integerwert in die entsprechende oktale Darstellung im Zahlensystem zur Basis 8. Dabei werden die Zahlen durch die Ziffern 0 bis 7 dargestellt. Der optionale Parameter 'y' gibt an, wie viele Stellen (1 bis 11) angegeben werden.

HEX$ wandelt eine Integerzahl in die entsprechende sedezimale (oder hexadezimale) Darstellung um. Dabei handelt es sich um ein Zahlensystem zur Basis 16 mit den Ziffern 0 bis 9 und den Buchstaben A bis F. Der wahlweise zweite Parameter 'y' gibt an, wie viele hexadezimale Ziffern dargestellt werden (maximal acht).

HEX$(x,n) entspricht RIGHT$(STRING$(n,48) + HEX$(x),n).

Beispiel:

```
x=32+15
a$=OCT$(16+7,4)
PRINT HEX$(x),a$,BIN$(1+4+16+64,8)
```

--> Schreibt 2F,0027 und 01010101 auf den Bildschirm.

VAL(a$)
VAL?(a$)

```
a$: sexp
```

VAL() wandelt eine Zeichenkette in eine Zahl. Trifft VAL() dabei auf ein Zeichen, das nicht mehr als Bestandteil einer Zahl interpretiert werden kann, so wird die Analyse des Strings abgebrochen. Die am String-Anfang stehende Zahl wird dann von VAL zurückgemeldet. Ist am String-Anfang keine Zahl zu finden, ergibt VAL Null.

Durch Angabe von &H(hex) oder &X(bin) oder &O(oct) können Zahlen in hexadezimaler, binärer und oktaler Schreibweise erkannt werden.

Außerdem können durch $ hex-Zahlen und % bin-Zahlen gekennzeichnet werden.

Mit VAL?() kann man überprüfen, wie viele Stellen einer Zahl mit VAL() umgewandelt werden. VAL?() ergibt Null, wenn keine gültige Zahl angegeben wird.

Beispiele:

```
a$=STR$(12345)
PRINT VAL(a$),VAL("-.123 abc 123"),VAL?("3.00 DM")
```

--> Gibt die Zahlen 12345, -.123 und 4 auf dem Bildschirm aus.

```
PRINT VAL("&H"+"AF")
```

--> Gibt die Zahl 175 aus.

```
PRINT VAL("$AA")
PRINT VAL("%10101010")
```

--> Gibt zweimal die Zahl 170 aus.

CVI(a$) **CVL**(a$) **CVS**(a$) **CVF**(a$) **CVD**(a$)
MKI$(x) **MKL**(x) **MKS**(x) **MKF**(x) **MKD**(x)

```
a$: sexp
x: aexp
```

Die Funktionen CVI, CVL, CVS, CVF und CVD wandeln Zeichenketten in Zahlen um. Im Gegensatz zu VAL/STR$ erfolgt keine Umwandlung in Klartext, sondern die interne Darstellung des Strings und der Zahlen ist entscheidend. Die einzelnen CVx-Funktionen haben folgende Wirkungen:

```
CVI wandelt eine 2-Byte-Zeichenkette in eine 16-Bit-Integerzahl.
CVL wandelt eine 4-Byte-Zeichenkette in eine 32-Bit-Integerzahl.
CVS wandelt eine 4-Byte-Zeichenkette, die eine Zahl im ST-BASIC
    kompatiblen Format enthält, in eine GFA-BASIC-Fließkommazahl um.
CVF wandelt eine 6-Byte-Zeichenkette im GFA-BASIC 1.0 und 2.0-Format
    in eine Fließkommazahl (im 3.0-Format) um.
CVD wandelt eine 8 Byte Zeichenkette in eine Fließkommazahl
    im GFA-BASIC 3.0-Format um.
```

MKI$, MKL$, MKS$, MKF$ und MKD$ sind die Umkehrfunktionen der CVx-Funktionen; ihre Bedeutung ergibt sich also aus der gerade aufgeführten Tabelle.

Dabei entspricht:

```
MKI$(x%)=CHR$(SHR(x%,8))+CHR$(x%)
MKL$(x%)=CHR$(SHR(x%,24))+CHR$(SHR(x%,16))+CHR$(SHR(x%,8))+CHR$(x%)
```

Das heißt, daß das höchstwertige Byte zuerst kommt.

Ein Anwendungsbeispiel ist das Lesen von Zahlenformaten anderer Programme sowie das platzsparende Speichern von Zahlen zum Beispiel in "R"-Files.

Beispiel:

```
a$=MKL$(1000)
PRINT CVL(a$),LEN(a$)
b$=MKD$(100.1)
PRINT CVD(b$),LEN(b$)
```

--> Schreibt die Zahlen 1000 und 4 sowie die Zahlen 100.1 und 8 auf den Bildschirm.

CINT(x)
CFLOAT(y)

```
x: aexp
y: iexp
```

Die Funktion CINT wandelt eine Fließkommazahl 'x' in einen gerundeten Integerwert. Analog wandelt CFLOAT einen Integerwert 'y' in eine Fließkommazahl. Diese Funktion wird normalerweise nicht benötigt und ist nur der Vollständigkeit halber aufgeführt. Im Compiler hat sie jedoch eine Bedeutung.

Beispiel:

```
a=1.2345
a%=10000
b%=CINT(a)
b=CFLOAT(a%)
PRINT b%,b
```

--> Gibt 1 und 10000 auf dem Bildschirm aus.

Pointer-Operationen

*

BYTE{}, CARD{}, INT{}, LONG{}, {}, FLOAT{},
SINGLE{}, DOUBLE{}, CHAR{}
xPEEK, xPOKE
V:, VARPTR
ARRPTR
ABSOLUTE

*x

```
x: svar oder ein Array-Name gefolgt von ()
```

Das Multiplikationszeichen kann auch als Pointersymbol (Zeigersymbol) dienen. In diesem Fall ergibt *x die Adresse, an der die Variable x im Speicher steht. Bei Zeichenketten ergibt *x$ die Adresse des String-Descriptors (ARRPTR(x$)). Der Ausdruck *x ist gleichbedeutend mit ARRPTR(x). Dies gilt auch für Arrays.

Eine besondere Bedeutung hat diese Variante der Pointeranwendung bei der indirekten Übergabe von Arrays und Variablen an Unterprogramme. Dazu kann man in Version 3.0 besser die Anweisung VAR benutzen.

Beispiel:

```
' indirekte Array-Übergabe
DIM a(3)
change(*a())
PRINT a(2)
'
PROCEDURE change(ptr%)
  SWAP *ptr%,x()
  ARRAYFILL x(),1
  SWAP *ptr%,x()
RETURN
```

--> Der Inhalt des Arrays a() wird verändert, ohne daß sein Name in der Prozedur change auftaucht. Es wird die Zahl 1 auf den Bildschirm geschrieben (siehe auch SWAP).

Oder in 3.0 besser:

```
DIM a(3)
change(a())
PRINT a(2)
'
PROCEDURE change(VAR x())
  ARRAYFILL x(),1
RETURN
```

PEEK(x) **DPEEK**(x) **LPEEK**(x)
POKE x,y **DPOKE** x,y **LPOKE** x,y
SPOKE x,y **SDPOKE** x,y **SLPOKE** x,y

```
x,y: iexp
```

Mit Hilfe der Funktion PEEK und des Befehls POKE und ihrer Varianten kann man Speicherstellen auslesen oder Werte in den Speicher schreiben. Die einzelnen Befehlsvarianten sind:

```
PEEK(x)      Liest ein Byte an der Adresse x.
DPEEK(x)     Liest zwei Bytes ab der Adresse x.
LPEEK(x)     Liest vier Bytes ab der Adresse x.

POKE x,y     Schreibt den Wert y als ein Byte an die Adresse x.
DPOKE x,y    Schreibt y als 2-Byte-Wert an die Adresse x.
LPOKE x,y    Schreibt y als 4-Byte-Wert an die Adresse x.
```

Bei der Verwendung von DPEEK, LPEEK, DPOKE und LPEEK muß darauf geachtet werden, daß nur gerade Adressen angegeben werden.

Varianten der POKE-Befehle arbeiten im Supervisormodus. So können auch geschützte Adressen (z.B. 0 bis 2047) erreicht werden. Die entsprechenden Befehle heißen SPOKE, SDPOKE und SLPOKE. Besonders im Supervisormodus ist Vorsicht geboten, da Manipulationen an geschützten Adressen schwerwiegende Folgen haben können. Die Peek-Funktionen arbeiten immer im Supervisormodus.

Beispiel:

```
LPOKE XBIOS(14,1)+6,0
```

--> Löscht den Tastaturpuffer (setzt Head- und Tail-Zeiger an den Pufferanfang).

Oder alternativ:

```
REPEAT
UNTIL INKEY$=""
```

--> Löscht den Tastaturpuffer durch zeichenweises Auslesen.

BYTE{x}
CARD{x}
INT{x}
LONG{x}
{x}
FLOAT{x}
SINGLE{x}
DOUBLE{x}
CHAR{x}

```
x: iexp
```

Mit Hilfe dieser Kommandos kann man bestimmte Variablentypen ab einer Adresse lesen oder an eine Adresse schreiben. Als Funktion (z.B. y=BYTE{x}) wird ab der Adresse x aus dem Speicher gelesen. Als Befehl (z.B. BYTE{x}=y) wird ab der Adresse x der Wert y in den Speicher geschrieben.

Bei der Verwendung von INT{}, CARD{}, LONG{}, {}, FLOAT{}, SINGLE{} und DOUBLE{} muß darauf geachtet werden, daß nur gerade Adressen angegeben werden, da sonst ein Adreßfehler (drei Bomben) ausgelöst wird.

Typ	Bedeutung
BYTE{x}	Liest/schreibt ein Byte.
CARD{x}	Liest/schreibt 2-Byte-Integer ohne Vorzeichen.
INT{x}	Liest/schreibt 2-Byte-Integer mit Vorzeichen.
LONG{x}	Liest/schreibt 4-Byte-Integerwert.
{x}	Liest/schreibt 4-Byte-Integerwert.
FLOAT{x}	Liest/schreibt 8-Byte-Fließkommavariable im GFA-BASIC 3.0-Format.
SINGLE{x}	Liest/schreibt 4-Byte-Fließkommavariable im IEEE-Single-Format.
DOUBLE{x}	Liest/schreibt eine 8-Byte-Fließkommavariable im IEEE-Double-Format.
CHAR{x}	Liest/schreibt einen mit einem Null-Byte endenden String. Besonders wichtig für die Kommunikation mit C-Routinen, GEM und GEMDOS.

Mit x% = LONG{adr%} wird der Variablen x% das Langwort, das an der Adresse adr% steht, zugewiesen und bei LONG{adr%} = x% wird an die Adresse adr% der Wert der Variablen x% als Langwort geschrieben.

Mit den SINGLE- und DOUBLE-Anweisungen ist es möglich, ein fremdes Zahlenformat zu lesen bzw. zu schreiben. Einige C-Compiler kennen das IEEE-DOUBLE-Format. Um z.B. eine GFA-BASIC Zahl in das SINGLE- oder DOUBLE-Format zu konvertieren und so hexadezimal auszugeben, kann man folgendes schreiben:

```
a$=SPACE$(4)
SINGLE{V:a$}=1.2345
PRINT HEX$(CVL(a$),8)
```

oder

```
a$=SPACE$(8)
DOUBLE{V:a$}=1.2345
PRINT HEX$({V:a$},8)'
PRINT HEX$({V:a$+4},8)
```

Einige der oben genannten Funktionen entsprechen in etwa den PEEK-Funktionen, LONG{x} (oder {x}) entspricht etwa LPEEK(x), arbeitet jedoch nicht im Supervisormodus. Deshalb ist {x} schneller als LPEEK. Bei Zugriff auf geschützte Speicherbereiche (0 bis 2047) wird ein Busfehler (zwei Bomben) ausgelöst und zeigt so Programmierfehler an.

Beispiele:

```
adr%=XBIOS(2)
t%=TIMER
FOR i%=1 TO 4000
  ~LPEEK(adr%)
NEXT i%
PRINT (TIMER-t%)/200
t%=TIMER
FOR i%=1 TO 4000
  ~{adr%}
NEXT i%
PRINT (TIMER-t%)/200
'
FLOAT{*x}=PI
```

--> Der erste Teil des Beispiels demonstriert, daß {} schneller als LPEEK arbeitet, der zweite Teil schreibt eine Fließkommazahl indirekt in eine Variable x.

```
BYTE{XBIOS(2)+4160}=&HFF
CARD{XBIOS(2)+4320}=&HFFFF
LONG{XBIOS(2)+4480}=&HFFFFFFFF
'
a$="Test"+CHR$(0)
PRINT CHAR{V:a$};
b$=SPACE$(5)
CHAR{V:b$}="wort"
PRINT b$,ASC(RIGHT$(b$))
```

--> Zuerst werden einige Werte direkt in den Bildschirmspeicher geschrieben und erscheinen dort als Striche. Dann werden sowohl a$ als auch b$ Strings zugewiesen, die mit einem Null-Byte enden, wobei a$ konventionell zugewiesen wird (über +CHR$()) und unkonventionell über CHAR{} ausgegeben, b$ wird dagegen über CHAR{} zugewiesen und bei PRINT mit dem nicht sichtbaren CHR$(0) ausgegeben.

VARPTR(x) **V:**x
ARRPTR(y) *y

```
x: Name einer Variablen beliebigen Typs
y: Name einer Variablen oder eines Arrays mit leeren Klammern
```

Mit ARRPTR(y) oder *y wird die Adresse einer Variablen y, bei Arrays oder Strings die Adresse des Deskriptors, bestimmt.

Mit VARPTR(x) oder V:x wird dagegen bei Strings die Adresse des Strings selber oder bei Arrays die Adresse eines Array-Elementes bestimmt.

Bei skalaren (nicht Feld oder String) Variablen sind VARPTR, V:, ARRPTR und * gleichbedeutend.

Beispiel:

```
DIM x%(10)
a$="Test"
PRINT ARRPTR(x%())),VARPTR(x%(0)),V:x%(1)
PRINT ARRPTR(a$),*a$,VARPTR(a$)
```

--> Gibt in der ersten Zeile die Adresse des Array-Deskriptors sowie die Adresse der ersten beiden Feldelemente von x%() aus. In der zweiten Zeile wird die Adresse des String-Deskriptors von a$ zweimal und die Adresse des ersten Bytes der Zeichenkette einmal ausgegeben.

ABSOLUTE x,y

```
x: eine Variable beliebigen Typs
y: iexp
```

Mit dem Befehl ABSOLUTE wird die Adresse (ARRPTR/*) einer Variablen verändert.

Beispiel:

```
ABSOLUTE y,*x
x=13
y=7
PRINT x,y,*x,*y
```

--> Hier wird die Variable y auf die Adresse von x umgeleitet, so daß am Ende beide Variablen mit gleichem Wert (7) und gleicher Adresse ausgeben werden.

Löschen und Vertauschen

CLEAR, CLR, ERASE
SWAP
SSORT, QSORT
INSERT, DELETE

CLEAR
CLR x [,y,...]
ERASE z1() [,z2(),...]

```
x,y: svar oder avar
z1,z2: Name beliebiger Arrays
```

Mit CLEAR werden alle Variablen und Felder gelöscht. Der Befehl kann nicht innerhalb von FOR-NEXT-Schleifen oder Unterprogrammen benutzt werden. Beim Starten eines Programms wird er automatisch ausgeführt, und er ist sonst nur zur Behandlung schwerwiegender Fehler mit RESUME x sinnvoll anwendbar.

Der Befehl CLR löscht die Variablen, die in der hinter ihm stehenden Liste aufgeführt sind. Arrays können mit CLR allerdings nicht gelöscht werden. ERASE löscht komplette Arrays, die nach diesem Kommando auch erneut dimensioniert werden können.

Im Unterschied zur Version 2.0 können auch mehrere Arrays mit einem ERASE-Befehl gelöscht werden, zum Beispiel ERASE x(),y().

Beispiel:

```
x=2
y=3
CLEAR
PRINT x,y
'
x=2
y=3
CLR x
PRINT x,y
'
DIM x(10)
PRINT FRE(0)
ERASE x()
PRINT FRE(0)
```

--> Schreibt dreimal die Zahl 0 und einmal die Zahl 3 auf den Bildschirm. Danach folgen zwei Zahlen, die zeigen, daß der durch DIM belegte Speicherplatz durch ERASE wieder freigegeben wird.

SWAP a,b
SWAP e(),f()
SWAP *c,d()

```
a,b: avar oder svar
c: Zeiger auf einen Array-Deskriptor
d,e,f: Name von Arrays
```

In seiner einfachsten Variante dient der Befehl SWAP dem Vertauschen von zwei Variablen desselben Typs (SWAP a,b). Er kann aber auch zum Vertauschen von zwei Feldern verwendet werden. Das Feldvertauschen geht sehr schnell, da nur die zugehörigen Deskriptoren vertauscht werden. So wird auch die Dimensionierung der beiden Arrays vertauscht. Arrays müssen hierzu nicht dimensioniert sein.

In der dritten Variante enthält c den Pointer auf einen Array-Deskriptor und d() ist der Name eines Arrays. Die ist vor allem bei der indirekten Übergabe von Feldern an Unterprogramme interessant (siehe zweites Beispiel).

Der Befehl SWAP sollte von der Funktion SWAP, die im Abschnitt über Bit-Operationen besprochen wird, klar unterschieden werden.

Beispiele:

```
x=1
y=2
PRINT x,y
SWAP x,y
PRINT x,y
```

--> Gibt nach 1 und 2 auch die Zahlen 2 und 1 aus.

```
DIM x(3)
change(*x())
PRINT x(2)
'
PROCEDURE change(adr%)
  SWAP *adr%,a()
  ARRAYFILL a(),1
  SWAP *adr%,a()
RETURN
```

--> Das Array x() wird in der Prozedur change mit Einsen gefüllt, ohne daß der Name x() im Unterprogramm auftaucht. So kann ein Unterprogramm für mehrere Felder verwendet werden. Dann wird, wie im Beispiel, die Deskriptoradresse übergeben und durch SWAP am Anang und am Ende werden diese Felder dann unter einem Namen angesprochen.

In der Version 3.0 empfiehlt sich hierbei die indirekte Feldübergabe als VAR-Parameter:

```
DIM x(3)
CHANGE (x())
PRINT x(2)
'
PROCEDURE change (VAR a())
  ARRAYFILL a(),1
RETURN
```

QSORT a(s) [,n] [,j%()]
QSORT x$(s) **WITH** i() [,n [,j%()]]

SSORT a(s) [,n] [,j%()]
SSORT x$(s) **WITH** i() [,n [,j%()]]

```
a(): beliebiges Array, auch String-Array
i(): Integer-Array (|,& oder %)
j%(): 4-Byte-Integer-Array
x$(): String-Array
n: iexp
s: +,- oder kein Zeichen
```

Mit Hilfe der Befehle SSORT und QSORT können die Elemente eines Arrays nach ihrer Größe sortiert werden. Bei SSORT wird dazu das Shellsort-Verfahren benutzt, bei QSORT das Quicksort-Verfahren.

In die Klammern, vor denen der Name des zu sortierenden Arrays steht, kann ein Plus- oder Minus-Symbol eingesetzt werden. Das Minus-Zeichen besagt, daß die Elemente des Arrays in absteigender Reihenfolge sortiert werden sollen. In diesem Fall steht das größte Element nach der Sortierung im nullten Array-Element. Das Plus-Zeichen führt dazu, daß das Array in aufsteigender Reihenfolge sortiert wird; im nullten Array-Element steht dann nach der Sortierung der kleinste Wert. Ohne Symbol wird aufsteigend sortiert (wie +).

Der Parameter 'n' gibt an, daß nur die ersten 'n' Elemente des Arrays sortiert werden sollen (falls OPTION BASE 0 gilt, sind dies die Elemente mit den Indizes 0 bis n-1, sonst 1 bis n).

Als dritter Parameter kann ein weiteres Integer-Array angegeben werden, das bei der Sortierung des ersten Arrays mitsortiert wird. Jede Vertauschung von Elementen im ersten Array wird auch im zweiten durchgeführt. Dies kann z.B. verwendet werden, wenn in einem Array ein Sortierschlüssel (z.B. Postleitzahl) und in mehreren weitere Informationen hierzu enthalten sind. Das mitsortierte Integerfeld enthält dabei z.B. Array-Indizes, "R"-Datei-Indizes, LOC-Werte für SEEK usw.

Bei der Sortierung von String-Arrays kann mit WITH ein Sortierkriterium in Form eines Arrays mit mindestens 256 Elementen angegeben werden. Ohne Angabe von WITH wird die normale ASCII-Tabelle als Sortierkriterium verwendet (siehe zweites Beispiel).

Beispiele:

```
DIM x%(20)
PRINT "Unsortiert:            ";
FOR i%=0 TO 10
  x%(i%)=RAND(9)+1
  PRINT x%(i%);"  ";
NEXT i%
PRINT
'
QSORT x%(),11
DIM index%(20)
PRINT "Absteigend sortiert:  ";
FOR i%=0 TO 10
  PRINT x%(i%);"  ";
  index%(i%)=i%
NEXT i%
PRINT
'
SSORT x%(-),11,index%()
PRINT "Aufsteigend sortiert: ";
FOR i%=0 TO 10
  PRINT x%(i%);"  ";
NEXT i%
PRINT
PRINT "Mitsortiertes Feld:   ";
FOR i%=0 TO 10
  PRINT index%(i%);"  ";
NEXT i%
```

--> Gibt ein unsortiertes Feld und zwei sortierte Reihen von Zufallszahlen aus. In einer vierten Reihe stehen die Werte eines mitsortierten Feldes.

```
DIM b|(256)
FOR i%=0 TO 255
  b|(i%)=ASC(UPPER$(CHR$(i%)))
NEXT%
FOR i%=1 TO 7
  READ a$,b$
  b|(ASC(a$))=ASC(b$)
NEXT i%
DATA Ä,A,Ö,O,Ü,U,ä,A,ö,O,ü,U,ß,S
DIM n$(3)
FOR i%=0 TO 3
  READ n$(i%)
NEXT i%
DATA Muhlmann,Müller,Überlauf,Unterlauf
QSORT n$(),4
FOR i%=0 TO 3
  PRINT n$(i%),
NEXT i%
PRINT
QSORT n$() WITH b|(),4
FOR i%=0 TO 3
  PRINT n$(i%)
NEXT i%
```

--> Sortiert zweimal das String-Array n$(), einmal ohne WITH, einmal mit. Die Angabe von WITH b|() führt hier dazu, daß die deutschen Umlaute wie die entsprechenden Nicht-Umlaute einsortiert werden und Kleinbuchstaben bei Großbuchstaben.

INSERT x(i) = y
DELETE x(i)

x: Name eines Arrays
i: iexp
y: aexp oder sexp, je nach Variablentyp des Arrays

Mit den Befehlen INSERT und DELETE kann ein Element in einem Array eingefügt oder gelöscht werden. INSERT fügt in das Array x an die Position i den Wert des Ausdrucks y ein. Dabei werden alle Elemente des Arrays, die einen Index größer i haben, um eine Position nach "oben" verschoben. Wenn ein Element z.B. vorher an der Position i + 3 stand, so steht es nach dem INSERT-Befehl an der Position i + 4. Das letzte Element des Arrays wird bei diesem Einfügevorgang gelöscht.

DELETE entfernt das i-te Element des Arrays x. Alle Array-Elemente, die einen Index größer i haben, werden dabei eine Position nach "unten" verschoben. Das letzte Element des Arrays wird dabei gleich 0 (bei Zeichenkettenfeldern zu einem Leer-String). Diese beiden Befehle eignen sich vor allem zur Verwaltung von Listen, in die ständig Elemente eingefügt und gelöscht werden müssen.

Beispiele:

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
INSERT x%(3)=33
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> Gibt die Zahlen 0, 1, 2, 33, 3 und 4 auf dem Bildschirm aus.

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
DELETE x%(3)
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> Gibt die Zahlen 0, 1, 2, 4, 5 und 0 auf dem Bildschirm aus.

Reservierte Variablen

FALSE, TRUE, PI
DATE$, TIME$, SETTIME
TIMER

FALSE
TRUE
PI

Die beiden Konstanten FALSE und TRUE enthalten Werte für logisch falsch (0) und logisch wahr (-1). Die Konstante PI enthält den Wert der Kreiszahl Pi.

Beispiel:

```
PRINT FALSE
IF TRUE
  PRINT PI
ENDIF
```

--> Schreibt die Zahlen 0 und 3.14159265359 auf den Bildschirm.

DATE$
TIME$
SETTIME zeit$,datum$
**DATE$** = datum$
**TIME$** = zeit$

```
zeit$,datum$: sexp
```

Die Funktion DATE$ ermittelt das Systemdatum im Format:

```
TT.MM.JJJJ  (Tag.Monat.Jahr) oder MM/TT/JJJJ (US-Format, siehe MODE)
```

TIME$ ermittelt die Systemuhrzeit. Hier ist das Format:

```
HH:MM:SS  (Stunden:Minuten:Sekunden)
```

TIME$ ändert sich dabei in Schritten von jeweils zwei Sekunden.

Mit dem Befehl SETTIME können Uhrzeit und Datum gestellt werden. Dabei müssen die Strings zeit$ und datum$ das Format haben, das gerade bei TIME$ und DATE$ angegeben wurde. Lediglich bei Jahresangaben zwischen 1980 und 2079 kann die Jahrhundertangabe entfallen. Wird SETTIME mit einem falschen Format der Strings zeit$ oder datum$ benutzt, so werden die aktuellen Werte nicht verändert. Außerdem können Datum und Zeit auch einzeln geändert werden über DATE$ = und TIME$ = .

Beispiel:

```
PRINT DATE$,TIME$
SETTIME "20:15:30","27.2.1988"
PRINT DATE$,TIME$
```

--> Schreibt das aktuelle Systemdatum und die Systemzeit auf den Bildschirm, setzt diese Variablen neu und gibt sie danach wieder aus.

TIMER

Mit der Funktion TIMER erhält man die Zeit seit dem Einschalten des Systems in zweihundertstel Sekunden.

Beispiel:

```
t%=TIMER
FOR i%=1 TO 2500
NEXT i%
PRINT (TIMER-t%)/200
```

--> Gibt die Zeit in Sekunden an, die für die Leerschleife benötigt wird.

Spezielles

LET, VOID, ~

Mit LET können auch Wertzuweisungen bei Variablennamen erfolgen, die gleich Befehlsworten sind. VOID ermöglicht die Benutzung einer Funktion, ohne daß ihr Rückgabewert weiterverwendet wird.

LET x=y

```
x: avar oder svar
y: aexp oder sexp
```

Mit Hilfe von LET kann man einer Variable den Wert eines Ausdrucks zuweisen. Der Ausdruck und die Variable müssen dabei entweder beide numerisch oder beide vom Zeichenkettentyp sein. Normalerweise ist die Benutzung der LET-Anweisung nicht notwendig. Sie diente in älteren BASIC-Dialekten dazu, Befehlsworte als Variablen verwenden zu können. GFA-BASIC erkennt aber gewöhnlich automatisch, ob ein Befehlswort als Variable verwendet wird.

Beispiel:

```
LET print=3
PRINT print
```

--> Gibt die Zahl 3 auf dem Bildschirm aus.

VOID fx
~ fi

```
fx: aexp
fi: iexp
```

In Programmiersprachen unterscheidet man normalerweise Befehle und Funktionen. Befehle führen dazu, daß irgendeine Tätigkeit durchgeführt wird. Bei Funktionen ermittelt diese Tätigkeit einen Wert, der dann zurückgemeldet wird. Dieser Rückgabewert kann dann Element eines Ausdrucks sein, mit PRINT ausgegeben werden oder auch mit einem Gleichheitszeichen einer Variablen zugewiesen werden, usw.

In vielen Fällen ist der Programmierer aber nicht an diesem Rückgabewert interessiert, sondern nur an der von der Funktion ausgeübten Tätigkeit. So meldet z.B. die Funktion INP(2) den Code einer gedrückten Taste zurück. Soll das Programm aber lediglich auf einen Tastendruck warten, so ist der Code der Taste, die das Warten beendet, nicht von Bedeutung. In einem solchen Fall kann GFA-BASIC mit VOID angewiesen werden, die Funktion auszuführen, aber den Rückgabewert zu vergessen. Im Gegensatz zu VOID wird bei Verwendung der Tilde '~' ein Integerausdruck berechnet und dann vergessen. Bei VOID wird z.B. bei INP(2) noch eine (überflüssige) Umwandlung in eine Fließkommazahl durchgeführt. Der Compiler erkennt diese überflüssigen Berechnungen selbst.

Beispiel:

```
VOID INP(2)
```

oder

```
~ INP(2)
```

--> Wartet dann nur auf einen Tastendruck ohne die, hier überflüssige, Zuweisung dummy=INP(2). Mit der Tilde spart der Interpreter auch noch die Umwandlung des Ergebnisses von INP(2) in einen Fließkommawert.

Speicherverwaltung

FRE
BMOVE
BASEPAGE, HIMEM
RESERVE
INLINE
MALLOC, MSHRINK, MFREE

FRE()
FRE(x)

x: aexp

Diese Funktion berechnet den freien Speicherplatz. Der Parameter x bleibt dabei unberücksichtigt. Darüber hinaus löst FRE(x) eine Garbage-Collection aus (Sammeln der noch benötigten String-Bereiche am oberen Speicherende). FRE() ergibt die Größe des Speicherbereichs, der ohne diese Garbage-Collection noch frei ist.

Beispiel:

```
frei%=FRE(0)
max%=frei%/3/4
DIM x%(max%)
PRINT frei%,max%
```

--> Dimensioniert ein Array so, daß es etwa ein Drittel des freien Speicherplatzes belegt. Ein 4-Byte-Integer-Feld belegt 4 Byte pro Element, deshalb die Division durch 4.

BMOVE ab_adr,an_adr,anzahl

```
ab_adr,an_adr,anzahl: iexp
```

Mit Hilfe des Befehls BMOVE können Speicherbereiche kopiert werden. Im Ausdruck ab_adr steht, ab welcher Adresse der zu kopierende Bereich steht. In an_adr steht, an welche Adresse die Kopie des Speicherbereichs abgelegt werden soll. In anzahl steht die Anzahl der zu kopierenden Bytes.

Der Befehl wird bei geraden Parametern deutlich schneller abgearbeitet als bei ungeraden. Er kann auch verwendet werden, wenn sich Orginal- und Kopierbereich überlagern.

Beispiel:

```
DIM screen2%(64000/4)
adr%=VARPTR(screen2%(0))
FOR i%=0 TO 300 STEP 100
  PBOX 0,i%,639,i%+50
NEXT i%
BMOVE XBIOS(2),adr%,32000
BMOVE XBIOS(2),adr%+32000,32000
HIDEM
REPEAT
  IF MOUSEY<>my%
    BMOVE adr%+my%*80,XBIOS(2),32000
    my%=MOUSEY
  ENDIF
UNTIL MOUSEK=2
```

--> Bei Bewegungen der Maus entlang der y-Achse wird ein Speicherbereich "durch den Bildschirm gescrollt".

BASEPAGE
HIMEM

In der Variablen BASEPAGE steht die Adresse der Basepage des GFA-BASIC-Interpreters. Die BASEPAGE ist ein 256 Byte langer Speicherbereich mit folgendem Aufbau:

Byte	Inhalt
0 bis 3	Adresse des TPA-Beginns (transient program area).
4 bis 7	Adresse des TPA-Endes plus 1.
8 bis 11	Adresse des TEXT-Segments des Programms.
12 bis 15	Länge des TEXT-Segments.
16 bis 19	Adresse des DATA-Segments.
20 bis 23	Länge des DATA-Segments.
24 bis 27	Adresse des BSS (block storage segment).
28 bis 31	Länge des BSS.
32 bis 35	Adresse der DTA (disk transfer adress).
36 bis 39	Adresse der Basepage des aufrufenden Prozesses.
40 bis 43	Reserviert.
44 bis 47	Adresse der Environment-Strings.
48 bis 127	Reserviert.
128 bis 255	Kommandozeile (das erste Byte gibt die Länge des Kommandotextes an).

In der Variable HIMEM steht, ab welcher Adresse der Speicher nicht mehr vom GFA-BASIC benutzt wird. Dies ist voreingestellt ein Wert, der 16384 Byte unterhalb des Bildschirmspeichers liegt.

Beispiel:

```
a%={BASEPAGE+&H2C}
DO
  a$=CHAR{a%}
  EXIT IF LEN(a$)=0
  PRINT a$
  ADD a%,SUCC(LEN(a$))      ! SUCC= +1
LOOP
```

--> Hier wird das komplette Environment ausgegeben.

RESERVE [n]

n: iexp

Mit Hilfe des RESERVE-Befehls kann angegeben werden, wie groß der von GFA-BASIC benutzte Speicherbereich sein soll. Wenn n positiv ist, so werden n Bytes für den Interpreter reserviert, der Rest wird freigegeben. Ist n negativ, so hat dies dieselbe Wirkung wie die beiden Befehle:

```
RESERVE
RESERVE FRE(0)-n
```

Ohne Angabe eines Parameters wird der Zustand wieder hergestellt, der beim Start des Interpreters vorlag.

Der Speicherbereich kann nur in Schritten von 256 Bytes reserviert werden. Der Befehl kann benutzt werden, um z.B. einen Speicherbereich für Daten oder Resource-Files freizugeben.

Wenn der Speicherbereich für GFA-BASIC mit RESERVE verkleinert wurde, so sollte man nicht vergessen, ihn später wieder zu vergrößern, da der verfügbare Platz sonst mit jedem Programmaufruf kleiner wird.

Beispiel:

```
RESERVE 2560
EXEC 0,"\WORDPLUS.PRG","",""
RESERVE
```

--> Reserviert 2560 Bytes für den Interpreter, lädt WORDPLUS.PRG (sofern vorhanden) und startet dieses Programm. Nach Abbruch von WORDPLUS.PRG wird der reservierte Speicherplatz wieder an das aufrufende Programm zurückgegeben.

INLINE adr,anz

```
adr: 4-Byte-Integervariable, keine Feldvariable
anz: Integerkonstante, die kleiner als 32700 ist
```

Hinter diesem Befehl ist kein Kommentar möglich, da intern genau an der Stelle, wo sonst der Kommentar steht, der gewünschte Speicherplatz reserviert wird. Anfänglich wird der Speicherplatz gelöscht (d.h. mit Null-Bytes gefüllt). Dieser Speicherbereich beginnt immer an einer geraden Adresse.

Beim Ausführen von INLINE wird diese Adresse in die Integervariable adr geschrieben. Beim Speichern oder Laden des Programms wird der reservierte Speicherbereich mitgespeichert bzw. mitgeladen.

Positioniert man den Cursor auf die Programmzeile, die den INLINE-Befehl enthält und drückt die Help-Taste, dann erscheint in der obersten Zeile des Editors eine Menüzeile mit den Einträgen LOAD, SAVE und CLEAR. Mit den Menüpunkten "LOAD" und "SAVE" ist es möglich Dateien in die Zeile mit dem INLINE-Befehl zu laden oder abzuspeichern. Dabei ist die Extension .INL voreingestellt. Der Menüpunkt "CLEAR" dient zum Löschen des reservierten Speicherbereiches. In diesen Speicherbereich können z.B. Bilder, Tabellen oder Assembler-Programme geladen werden.

Beispiel: Siehe Beispiel zu C: im Abschnitt über Systemroutinen.

MALLOC(x)
MFREE(y)
MSHRINK(y,z)

```
x,y,z: iexp
```

Die Funktion MALLOC (GEMDOS 72) dient der Reservierung (Allokierung) von Speicherbereichen. Ist ihr Parameter x gleich -1, so meldet die Funktion die Länge des größten zusammenhängenden freien Speicherbereichs. In diesem Fall ist MALLOC also eine Nachfragefunktion.

Ist x eine positive Zahl, so bedeutet dies, daß x Bytes reserviert werden sollen. In diesem Fall gibt MALLOC die Anfangsadresse des reservierten Speicherbereichs zurück. Trat beim Reservierungsversuch ein Fehler auf, so wird 0 zurückgemeldet.

Wenn größere Speicherbereiche allokiert werden sollen, so muß zunächst der von GFA-BASIC benutzte Speicherplatz mit RESERVE verkleinert werden. Allokierte Speicherbereiche sollten unbedingt vor der Beendigung des Programms wieder freigegeben werden. Dies wird allerdings beim Verlassen des Interpreters automatisch durchgeführt.

MFREE (GEMDOS 73) gibt den mit MALLOC reservierten Speicherplatz wieder frei. Dabei steht in y die Anfangsadresse des freizugebenden Speicherblocks, also ein bei MALLOC erhaltener Rückgabewert. Der zurückgemeldete Wert ist 0, wenn die Freigabe ohne Fehler abgeschlossen werden konnte, oder eine negative Fehlernummer.

MSHRINK (GEMDOS 74) verkleinert einen reservierten Speicherbereich, der vorher mit MALLOC allokiert wurde. Der Parameter y enthält dabei die Adresse des reservierten Speicherbereichs, die bei MALLOC zurückgemeldet wurde. In z steht die neue Sollänge des zu verkürzenden Speicherblocks. Die Funktion MSHRINK ergibt 0 bei korrekter Durchführung der Verkleinerung, -40, wenn als y eine falsche Adresse angegeben wurde und -67, wenn die neue Sollänge größer ist als die aktuelle Länge.

Wichtig ist, daß bei MFREE und MSHRINK nie ein fehlerhafter Pointer übergeben wird.

Beispiel:

```
RESERVE 1000
PRINT MALLOC(-1)
adr%=MALLOC(60000)
PRINT adr%
IF adr%>0
  x%=MSHRINK(adr%,30000)
  y%=MFREE(adr%)
ENDIF
RESERVE
```

--> Gibt die Größe des größten freien zusammenhängenden Speicherbereichs und danach die Adresse eines reservierten Speicherbereichs von 60000 Bytes, verkleinert diesen (wenn erfolgreich reserviert) auf 30000 und gibt ihn dann gänzlich frei.

3 - Operatoren

Operatoren sind Elemente einer Programmiersprache, die der Verknüpfung und dem Vergleich von numerischen, logischen und Zeichenkettenausdrücken dienen. In diesem Abschnitt werden die Operatoren von GFA-BASIC 3.0 in folgender Reihenfolge vorgestellt:

Im ersten Abschnitt werden die numerischen Operatoren besprochen (+, -, *, /, ^, DIV, \, MOD). Sie verknüpfen jeweils zwei numerische Ausdrücke und erzeugen eine Zahl, die z.B. mit dem Gleichheitszeichen einer Variablen zugewiesen werden kann. Die Operatoren + und - haben noch eine zweite Funktion; sie können als Vorzeichen verwendet werden.

Der zweite Abschnitt beschäftigt sich mit den logischen Operatoren (AND, OR, XOR, NOT, IMP, EQV). Sie verknüpfen zwei logische Ausdrücke und liefern einen Wahrheitswert (wahr oder falsch) zurück. Hier nimmt der Operator NOT eine Sonderstellung ein. Er wirkt nur auf einen logischen Ausdruck und negiert dessen Wahrheitswert (aus wahr wird falsch und umgekehrt).

Abschnitt 3 hat den Zeichenkettenoperator zum Thema. Dieser Operator ist das Plus-Zeichen (+), das zwei Zeichenkettenausdrücke verknüpft.

Der nächste Abschnitt beschreibt die Vergleichsoperatoren. Mit ihrer Hilfe können zwei numerische oder zwei Zeichenkettenausdrücke verglichen werden. Als Ergebnis erhält man einen Wahrheitswert (wahr oder falsch).

Im letzten Abschnitt geht es um die Reihenfolge, in der Operatoren abgearbeitet werden, wenn mehrere von ihnen in einem numerischen oder logischen Ausdruck auftauchen (Operatorenhierarchie). Dort werden auch die Klammersymbole "(" und ")" aufgeführt, mit deren Hilfe die Reihenfolge der Abarbeitung verändert werden kann.

Arithmetische Operatoren

+ - * / ^ DIV \ MOD
+ -

Die arithmetischen Operatoren +, -, *, /, ^, DIV, \ und MOD verknüpfen zwei numerische Ausdrücke und erzeugen eine Zahl. Diese Zahl kann selbst wiederum Bestandteil eines numerischen oder logischen Ausdrucks sein, einer Variablen zugewiesen oder, z.B. mit PRINT, ausgegeben werden. Die Operatoren + und - finden auch als Vorzeichen Verwendung.

```
x+y        Erzeugt die Summe der Zahlen x und y (Addition).
x-y        Erzeugt die Differenz x minus y (Subtraktion).

x*y        Erzeugt das Produkt x mal y (Multiplikation).
x/y        Erzeugt den Quotienten x geteilt durch y (Division).

x^y        Erzeugt die Potenz einer Zahl mit x als Basis und y als
           Exponent (Potenzierung).

x DIV y    Ergibt den ganzzahligen Anteil des Divisonsergebnisses
x \ y      von x geteilt durch y (Ganzzahldivision).

x MOD y    Ergibt den Divisionsrest der Division x geteilt durch
           y (Modulo-Berechnung).
```

Für DIV und MOD gelten folgende Zusammenhänge:

```
x DIV y = TRUNC(x/y)
x MOD y = x-y*TRUNC(x/y)
```

Beispiel:

```
13 DIV 4   erzeugt 3
13 MOD 4   erzeugt 1
```

+x Versieht den Wert x mit einem positiven Vorzeichen, ergibt also x.

-x Versieht den Wert x mit einem negativen Vorzeichen. Die erzeugte Zahl hat also ein negatives Vorzeichen, wenn x positiv war und ein positives Vorzeichen, wenn x negativ war.

Logische Operatoren

AND OR XOR NOT IMP EQV

Diese logischen Operatoren arbeiten auf Bit-Ebene für 32-Bit-Integerwerte.

Logische Operatoren verknüpfen zwei logische Ausdrücke und erzeugen einen Wahrheitswert (wahr oder falsch). Eine Ausnahme bildet der Operator NOT, der den Wahrheitswert des hinter ihm stehenden Ausdrucks negiert.

Der numerische Wert für logisch falsch (FALSE) ist 0. Jeder andere Zahlenwert wird als logisch wahr interpretiert. Die Variable TRUE hat dabei den Wert -1. Alle logischen Operatoren können auch auf numerische Ausdrücke angewendet werden. In diesem Fall werden die logischen Operationen bitweise ausgeführt.

Die Wirkung logischer Operatoren läßt sich am besten mit sogenannten Wahrheitstabellen beschreiben. In diesen Tabellen stehen in den ersten Spalten die Wahrheitswerte der verknüpften Ausdrücke (Operanden) und in der letzten Spalte der erzeugte logische Wert.

NOT x

```
x: iexp
```

Der Operator NOT (Negation) negiert den hinter ihm stehenden logischen Ausdruck. Er ist der einzige logische Operator, der nur ein Argument hat.

Er ändert jedes einzelne Bit seines Arguments.

x	NOT x
w	f
f	w

Beispiele:

```
PRINT NOT FALSE
PRINT NOT TRUE
PRINT NOT 0
```

--> Es erscheinen die Zahlen -1, 0 und -1 auf dem Bildschirm.

```
x=1
PRINT BIN$(x,2)
PRINT BIN$(NOT x,2)
--> 01
    10 erscheint auf dem Bildschirm.

x%=17
PRINT BIN$(x%,8),x%
PRINT BIN$(NOT x%,8),NOT x%
```

--> Ausgabe:
00010001 17
11101110 -18

x **AND** y

```
x,y: iexp
```

Der logische Operator AND (Konjunktion) prüft, ob zwei logische Ausdrücke x und y beide wahr sind. Nur in diesem Fall erzeugt er den Wert TRUE (logisch wahr, -1). Ist einer der beiden oder sind beide logischen Ausdrücke falsch, so erzeugt auch AND ein logisches falsch.

Mit AND wird jedes der 32 Bits seiner Argumente verknüpft.

x	y	x AND y
w	w	w
w	f	f
f	w	f
f	f	f

Beispiele:

```
PRINT TRUE AND -1
PRINT FALSE AND TRUE
```

--> Auf dem Bildschirm erscheinen die Zahlen -1 und 0.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x AND y,4),x AND y
```

--> Ausgabe:
0011
1010
0010 2

x **OR** y

```
x,y: iexp
```

Der Befehl OR (Disjunktion) prüft, ob mindestens einer von zwei logischen Ausdrücken x und y wahr ist. In diesem Fall wird logisch wahr erzeugt. Nur wenn x und y beide logisch falsch sind, ergibt auch x OR y logisch falsch. Im Unterschied zu XOR wird für den Fall, daß x und y beide wahr sind, auch ein logisches wahr erzeugt.

OR bedeutet, daß eines oder beide Argumente wahr sind. Auch OR arbeitet auf Bit-Ebene.

x	y	x OR y
w	w	w
w	f	w
f	w	w
f	f	f

Beispiele:

```
PRINT TRUE OR -1
PRINT FALSE OR TRUE
PRINT 0 OR FALSE
```

--> Es erscheinen die Zahlen -1, -1 und 0 auf dem Bildschirm.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x OR y,4),x OR y
--> 0011
    1010
    1011 und die Zahl 11 erscheinen auf dem Bildschirm.
```

x **XOR** y

x,y: iexp

Dieser Operator prüft, ob einer von zwei logischen Ausdrücken x und y logisch wahr ist. In diesem Fall ergibt auch x XOR y logisch wahr. Sind x und y beide logisch wahr oder beide logisch falsch, so ergibt XOR logisch falsch.

Der Unterschied zu OR besteht darin, daß TRUE OR TRUE wahr und TRUE XOR TRUE falsch ist (XOR = exklusive Disjunktion).

XOR ergibt dann (und nur dann) wahr, wenn genau in einem seiner Argumente das entsprechende Bit gesetzt ist. Auch XOR arbeitet für jedes der 32 Bits seiner Argumente einzeln.

x	y	x XOR y
w	w	f
w	f	w
f	w	w
f	f	f

Beispiele:

```
PRINT FALSE XOR -1
PRINT -1 XOR 1
PRINT 0 XOR FALSE
```

--> Es erscheinen die Zahlen -1, 0 und 0 auf dem Monitor.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x XOR y,4),x XOR y
```

--> Ausgabe:

```
0011
1010
1001      9
```

x **IMP** y

x,y: iexp

Der Operator IMP (Implikation) entspricht einer logischen Folgerung. Eine solche logische Folgerung kann nur dann falsch gewesen sein, wenn aus einer wahren Aussage etwas Falsches folgt. Dementsprechend ist x IMP y nur dann falsch, wenn x wahr und y falsch ist.

IMP arbeitet auf Bit-Ebene. Im Gegensatz zu AND, OR, XOR und EQV ist hier die Reihenfolge der Argumente wichtig.

x	y	x IMP y
w	w	w
w	f	f
f	w	w
f	f	w

Beispiele:

```
PRINT TRUE IMP -1
PRINT 0 IMP FALSE
PRINT TRUE IMP 0
```

--> Es erscheinen die Zahlen -1, -1 und 0 auf dem Bildschirm.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x IMP y,4)
```

--> Ausgabe:
0011
1010
1110

x **EQV** y

x,y: iexp

Der Operator EQV (Äquivalenz) erzeugt dann logisch wahr, wenn die logischen Ausdrücke x und y den gleichen Wahrheitswert haben.

EQV arbeitet auf Bitebene und setzt die Bits, die in beiden Argumenten gleich sind. Dies ist genau das Gegenteil von XOR, so daß (x EQV y) gleichbedeutend ist mit (NOT x XOR y).

A	B	A EQV B
w	w	w
w	f	f
f	w	f
f	f	w

Beispiel:

```
PRINT TRUE EQV FALSE
PRINT FALSE EQV FALSE
```

--> Auf dem Bildschirm erscheinen die Zahlen 0 und -1.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x EQV y,4)
```

--> Ausgabe:
0011
1010
0110

Zeichenkettenoperator

a$+b$

```
a$,b$: sexp
```

Mit dem Operator + können auch Zeichenketten verknüpft werden. Das Ergebnis ist eine Zeichenkette, die a$ und b$ enthält.

Beispiel:

```
a$="GFA-"
PRINT a$+"BASIC"
```

--> Auf dem Bildschirm erscheint der Text 'GFA-BASIC'.

Vergleichsoperatoren

= == >= <= <>

Mit Hilfe der Vergleichsoperatoren können numerische, logische und Zeichenkettenausdrücke miteinander verglichen werden. Das Ergebnis dieses Vergleichs ist immer ein logischer Wert (wahr = -1 oder falsch = 0). Eine Ausnahme bildet der Operator = =. Mit ihm können keine String-Ausdrücke verglichen werden.

x=y

```
x,y: exp
```

Der Operator = vergleicht zwei numerische oder Zeichenkettenausdrücke auf Gleichheit. Wenn die beiden Ausdrücke gleich sind, so wird logisch wahr erzeugt, andernfalls logisch falsch.

Beispiel:

```
x=6
IF 2=x/3
  PRINT "Ok"
ENDIF
PRINT 2=x/3
```

--> Es erscheint Ok und -1 auf dem Bildschirm.

x= =y

```
x,y: aexp
```

Vergleicht zwei numerische Ausdrücke auf ungefähre Gleichheit. Bei diesem Vergleich werden achteinhalb Nachkommastellen (28 Bit der Mantisse der Fließkommazahl) für den Vergleich herangezogen. Die Verwendung von = = ist dann sinnvoll, wenn Fließkommazahlen verwendet werden, bei denen Rundungsungenauigkeiten auftreten können.

Beispiele:

```
PRINT 1.0000000001=1
PRINT 1.0000000001==1
```

--> Es erscheinen die Zahlen 0 und -1 auf dem Bildschirm.

```
a=SINQ(77)
b=SIN(RAD(77))
PRINT a=b
PRINT a==b
```

--> Es erscheinen die Zahlen 0 (logisch falsch) und -1 (logisch wahr).

x<y
x>y
x< =y
x> =y

```
x,y: exp
```

Diese Operatoren dienen dem Größenvergleich von numerischen und String-Ausdrücken. Bei numerischen Ausdrücken werden die Zahlen verglichen, die sich aus ihnen berechnen lassen.

Bei String-Ausdrücken richtet sich der Vergleich nach dem ASCII-Code der Zeichen. Der String "ABC" wird als die Zahlfolge 65, 66 und 67 behandelt. Lautet die Aussage "ABC" > "AAA", so werden zunächst die ersten beiden Zeichen verglichen, die beide den ASCII-Code 65 haben. Nun wird das nächste Zeichen zum Vergleich herangezogen. Da B (ASCII-Code 66) einen höheren Code als A hat, wird B als "größer" aufgefaßt. An dieser Stelle wird der Vergleich der beiden Strings abgebrochen, und das Ergebnis der Aussage ist logisch wahr, "ABC" ist also größer als "AAA".

Ein Sonderfall des String-Vergleichs tritt ein, wenn einer der beiden Strings endet, bevor ungleiche Zeichen entdeckt wurden. Ein Beispiel wäre: "AA" > "A". Diese Aussage ist logisch wahr, da ein nicht existierendes Zeichen als "kleinstes Zeichen" gewertet wird. Selbst "A" + Chr$(0) > "A" ist logisch wahr.

Nun zu den einzelnen Operatoren:

```
 x>y ist wahr, wenn x größer als y ist.
 x<y ist wahr, wenn x kleiner als y ist.
x>=y ist wahr, wenn x größer oder gleich y ist.
x<=y ist wahr, wenn x kleiner oder gleich y ist.
```

Die Schreibweisen x< =y und x= <y sowie x> =y und x= >y sind gleichwertig. Auch wird x> <y in x< >y umgewandelt.

Beispiel:

```
PRINT "AAA">"aaa"
PRINT -1<=4-5
```

--> Es erscheinen die Zahlen 0 und -1 auf dem Bildschirm.

x< >y

```
x,y: exp
```

Dieser Operator prüft, ob zwei numerische oder String-Ausdrücke ungleich sind. Wenn dies der Fall ist, so ist die Aussage x< >y logisch wahr. Sind x und y gleich, so wird x< >y logisch falsch. Die Schreibweisen x< >y und x> <y sind gleichwertig.

Beispiel:

```
PRINT "Test"<>"test"
PRINT -1<>4-5
```

--> Es erscheinen die Zahlen -1 und 0 auf dem Bildschirm.

Zuweisungsoperator

```
x=y
 x: var
 y: exp
```

Das Gleichheitszeichen = kann nicht nur als Vergleichsoperator benutzt werden, sondern auch als Zuweisungsoperator. Dabei wird der Wert des rechts vom Gleichheitszeichen stehenden Ausdrucks y bestimmt und dieser Wert der links stehenden Variable x zugewiesen.

Numerische Ausdrücke können nur numerischen Variablen zugewiesen werden, Zeichenkettenausdrücke können nur String-Ausdrücken zugewiesen werden.

Zur Zuweisung gibt es auch noch den gleichwertigen Befehl LET var = exp (vgl. LET), der auch die Zuweisung an Variablen wie z.B. let oder rem erlaubt.

Beispiel:

```
x=LEN("TEST")+3
a$="GF"+CHR$(65)
PRINT x,a$
```

--> Auf dem Bildschirm erscheint 7 und 'GFA'.

Operatorenhierarchie

Wenn sich verschiedene Operatoren in einem Ausdruck befinden, so werden sie in einer bestimmten Reihenfolge abgearbeitet. Diese Reihenfolge ist abhängig von ihrem Platz in der sogenannten Operatorenhierarchie. Die Operatoren, die in dieser Hierarchie ganz oben stehen, werden als erste bearbeitet.

Die Hierarchie lautet:

( )	Klammern
+	Zeichenketten-Addition
= < > => <= <>	Zeichenketten-Vergleich
+ -	Vorzeichen
^	Potenzierung
* /	Multiplikation, Division
DIV MOD	Ganzzahlige Division und Modulo
+ -	Addition, Subtraktion
= == < <= >= > <>	Vergleichsoperatoren
AND OR XOR IMP EQV	Logische Operatoren
NOT	Negation

Mit Hilfe der Klammern ist es möglich, daß Operatoren, die in der Hierarchie niedrig stehen, dennoch zuerst abgearbeitet werden. In der Rangfolge der Operatoren ist auch die bekannte Regel "Punkt- vor Strichrechnung" zu erkennen.

Beispiel:

```
PRINT 2+4*3
PRINT (2+4)*3
PRINT 2+(4*3)
PRINT 3*2^2
```

--> Auf dem Bildschirm erscheinen die Zahlen 14, 18, 14 und 12.

4 - Numerische Funktionen

Mathematische Funktionen

ABS, SGN
ODD, EVEN
INT, TRUNC, FIX, FRAC, ROUND
MAX, MIN
SQR
EXP, LOG, LOG(10)
SIN, COS, TAN, ASIN, ACOS, ATN, DEG, RAD
SINQ, COSQ

Es gibt numerische Funktionen für folgende Aufgaben: Die numerischen Funktionen ABS und SGN liefern den Absolutbetrag und das Vorzeichen eines numerischen Ausdrucks. ODD und EVEN prüfen, ob eine Zahl ungerade oder gerade ist. INT, TRUNC, FIX und FRAC ergeben die Vor- oder Nachkommastellen von numerischen Ausdrücken. ROUND rundet einen Ausdruck.

MAX und MIN liefern den größten oder den kleinsten von mehreren numerischen Ausdrücken und SQR die Wurzel eines Ausdrucks. Die trigonometrischen Funktionen sind ASIN, ACOS, SIN, COS, TAN und ATN. EXP und LOG berechnen Potenzen und Logarithmen zur Eulerschen Zahl. LOG10 berechnet den Logarithmus zur Basis 10.

DEG und RAD dienen der Umwandlung von Grad- in Bogenmaß. RND, RANDOM, RAND und RANDOMIZE dienen der Erzeugung von Zufallszahlen.

ABS(x)
SGN(x)

x: aexp

Die Funktion ABS liefert den Absolutbetrag eines numerischen Ausdrucks. Dies führt zu folgenden Rückgabewerten:

x ist		ABS(x) ist dann
Negativ	-->	-x
Gleich 0	-->	0
Positiv	-->	x

Mit der Funktion SGN kann man ermitteln, welches Vorzeichen ein numerischer Ausdruck hat. Es gilt:

x ist		SGN(x) ist dann
Negativ	-->	-1
Gleich 0	-->	0
Positiv	-->	1

Beispiel:

```
x=-2
y=ABS(x)
PRINT SGN(x),ABS(5-3),SGN(ABS(x*3))  !ABS(x*3) ist gleich 6
```

--> Auf dem Bildschirm erscheinen die Zahlen -1, 2 und 1.

ODD(x)
EVEN(x)

x: aexp

Diese beiden Funktionen prüfen, ob der numerische Ausdruck x gerade oder ungerade ist. ODD liefert den Wert -1 (TRUE), wenn x ungerade ist und 0 (FALSE), wenn x gerade ist. EVEN ergibt -1 für ein gerades x und 0 für ein ungerades x. 0 wird wie eine gerade Zahl behandelt.

x ist		ODD(x) ist dann	EVEN(x) ist dann
Gerade	-->	0 (FALSE)	-1 (TRUE)
Ungerade	-->	-1 (TRUE)	0 (FALSE)
Gleich 0	-->	0 (FALSE)	-1 (TRUE)

Beispiel:

```
x=2
PRINT ODD(x),EVEN(-2),ODD(3*5), EVEN(-3*x)
```

--> Es erscheinen die Zahlen 0, -1, -1 und 0 auf dem Bildschirm.

INT(x)
TRUNC(x)
FIX(x)
FRAC(x)

```
x: aexp
```

Diese Funktionen liefern die Vor- und Nachkommastellen von numerischen Ausdrücken. INT, TRUNC (und die mit TRUNC identische Funktion FIX) liefern eine ganze Zahl zurück. Die Funktion TRUNC schneidet die Nachkommastellen von x ab. INT liefert die größte ganze Zahl zurück, die kleiner oder gleich x ist. Der Unterschied zwischen TRUNC (FIX) und INT ist bei positiven x-Werten nicht zu erkennen. Bei einem negativen, nachkommabehafteten x entsteht jedoch ein Unterschied. So würde TRUNC bei einem x von -1.2 die Nachkommastelle entfernen und erhält den Wert -1. INT dagegen sucht die nächste ganze Zahl, die kleiner als -1.2 ist und findet dadurch -2 als Ergebnis.

FRAC liefert die Nachkommastellen von x, entfernt also die Vorkommastellen. Der durch FRAC erhaltene Wert hat dasselbe Vorzeichen wie x. FRAC ist komplementär zu TRUNC und nicht zu INT. Es gilt:

```
x = TRUNC(x) + FRAC(x)
```

Ersetzt man TRUNC durch INT, so stimmt die Gleichung nicht mehr im negativen Zahlenbereich (z.B. für y gleich -1.2).

Beispiel:

```
x=-1.4
y=TRUNC(1.3)
PRINT y,INT(x),FIX(3*x),FRAC(x-3)
```

--> Auf dem Bildschirm erscheinen die Zahlen 1, -2, -4 und -0.4.

ROUND(x[,n])

```
x: aexp
n: iexp
```

Die Funktion ROUND(x) liefert einen gerundeten Wert. Die Variante ROUND(x,n) rundet den Ausdruck 'x' auf 'n' Nachkommastellen. Ist 'n' gleich Null, wird wie bei ROUND(x) auf ganze Zahlen gerundet. Ist 'n' negativ, wird vor dem Komma gerundet. So ergibt ROUND(155,-1) die Zahl 160 (vgl. auch CINT() = Runden mit Integer-Ergebnis).

Beispiele:

```
y=ROUND(-1.2)
PRINT y,ROUND(1.7)
```

--> Gibt -1 und 2 auf dem Bildschirm aus.

```
FOR i%=-5 TO 5
  PRINT i%,ROUND(PI*100,i%)
NEXT i%
```

--> Gibt die Laufvariable und den zugehörigen, formatierten Ausdruck auf dem Bildschirm aus. Dazu wird die Kreiszahl Pi mit 100 multipliziert und auf i% Stellen gerundet ausgegeben. Ist i% negativ, wird vor dem Komma gerundet.

MIN(x [,y,z,...])
MIN(x$ [,y$,z$,...])
MAX(x [,y,z,...])
MAX(x$ [,y$,z$,...])

```
x,y,z: aexp
x$,y$,z$: sexp
```

Die Funktion MAX liefert den größten der numerischen Ausdrücke x,y,z,... , die in der Parameterliste dieser Funktion aufgeführt sind. Analog dazu bestimmt MIN den kleinsten Ausdruck. Außerdem können die Funktionen MIN und MAX ebenso auf Zeichenkettenausdrücke angewandt werden.

Beispiel:

```
x=3
y=MAX(3,5,5-4)
PRINT MIN(x,y),MAX(-1,x*2)
```

--> Auf dem Bildschirm erscheinen die Zahlen 3 und 4.

SQR(x)

```
x: aexp
```

Meldet die Quadratwurzel des numerischen Ausdrucks x zurück. Wenn der Ausdruck x kleiner als Null ist, erzeugt dies eine Fehlermeldung.

Beispiele:

```
x=9
y=SQR(x)
PRINT y,SQR(4*4)
```

--> Es erscheinen die Zahlen 3 und 4 auf dem Bildschirm.

```
PRINT SQR(SQR(16))
PRINT SQR(-2)
```

--> Es erscheint die Zahl 2 und dann eine Fehlermeldung.

EXP(x)
LOG(x)
LOG10(x)

```
x: aexp
```

EXP berechnet die x-te Potenz zur Basis der Euler'schen Zahl (e=2.1782818284...) und LOG bildet die Umkehrfunktion dazu, bestimmt also den Logarithmus von x zur Basis e (natürlicher Logarithmus). Analog dazu gibt LOG10 den Logarithmus von x zur Basis 10 zurück (dekadischer Logarithmus).

Der numerische Ausdruck x muß bei den Logarithmusfunktionen größer als Null sein, sonst erscheint eine Fehlermeldung.

Beispiel:

```
x=2
y=EXP(2)
PRINT y,LOG10(2*5),LOG(x)
```

--> Auf dem Bildschirm erscheint 7.389056098931, 1 und 0.6931471805599.

SIN(x)
COS(x)
TAN(x)

ASIN(x)
ACOS(x)
ATN(x)

DEG(x)
RAD(grad)
SINQ(grad)
COSQ(grad)

```
x,grad: aexp
```

Dies sind die trigonometrischen Funktionen. Der numerische Ausdruck x wird als Bogenmaß interpretiert. Die Funktionen berechnen:

```
SIN  --> Sinus
COS  --> Cosinus
TAN  --> Tangens
ASIN --> Arcussinus
ACOS --> Arcuscosinus
ATN  --> Arcustangens
```

Um einen Winkel vom Bogen- ins Gradmaß umzurechnen, benutzt man die Funktion DEG() bzw. deren Umkehrfunktion RAD(). DEG(x) entspricht dabei (x * 180 / PI).

Die Funktionen SINQ und COSQ liefern interpolierte Sinus- bzw. Cosinuswerte, dazu wird GFA-BASIC-intern eine Tabelle mit den Sinuswerten in Grad-Schritten benutzt. Entsprechend dem Wert des Funktionsarguments grad werden die Zwischenwerte in 1/16-Grad-Schritten linear interpoliert. Für die grafische Darstellung auf dem Bildschirm ist diese Genauigkeit ausreichend und von den mit SIN bzw. COS berechneten Werten nicht zu unterscheiden, aber wesentlich schneller (etwa um den Faktor 10).

Im Unterschied SIN und COS erwarten SINQ und COSQ ihre Parameter im Gradmaß.

`SINQ(grad)`	entspricht	`SIN(RAD(grad))`
`COSQ(grad)`	entspricht	`COS(RAD(grad))`

Beispiele:

```
x=90
y=COS(x*PI/180)
z=270*PI/180
PRINT y,SIN(z),TAN(45),ATN(1/2)
```

--> Es erscheinen die Zahlen 1,-1,1.619775190544 und 0.4636476090008.

```
alpha%=30
PRINT SINQ(alpha%)
```

--> Gibt 0.5 auf dem Bildschirm aus.

Zufallszahlengenerator

RND [(x)]
RANDOM(x)
RAND(y)
RANDOMIZE y

```
x: aexp
y: iexp
```

Diese Gruppe von Kommandos dient der Erzeugung von Zufallszahlen. RND erzeugt eine Zufallszahl zwischen 0 und 1 (inklusive 0, exklusive 1). Der optionale Parameter x hat keine Bedeutung.

RANDOM erzeugt eine ganzzahlige Zufallszahl zwischen 0 und x (inklusive 0, exklusive x). Der numerische Ausdruck x muß nicht ganzzahlig sein, wenn nicht alle Zahlen mit gleicher Wahrscheinlichkeit erscheinen sollen.

RAND erzeugt eine zufällige 16-Bit-Integerzahl im Bereich von 0 bis y-1. Es werden 16-Bit von y ausgewertet.

Der Befehl RANDOMIZE initialisiert den Zufallszahlengenerator mit dem Wert 'y'. So erhält man, wenn man den Zufallszahlengenerator mehrmals mit demselben 'y' initialisiert, danach immer dieselbe Folge von Zufallszahlen. Zu Beginn jedes Programmlaufs wird der Zufallszahlengenerator mit einer "zufällig" ausgewählten Zahl initialisiert. Ohne die Verwendung von RANDOMIZE erhält man also nach jedem Programmstart andere Zufallszahlen mit RND, RANDOM oder RAND.

Zur Initialisierung des Zufallszahlengenerators kann man RANDOMIZE auch ohne Parameter oder RANDOMIZE 0 benutzen.

Beispiele:

```
x=RND
PRINT x,RND(2)
```

--> Zwei Zufallszahlen zwischen 0 und 1 erscheinen auf dem Monitor.

```
x=RANDOM(2)
y=RAND(4)
PRINT x,y,RAND(x),RANDOM(3*x)
```

--> Vier ganzzahlige Zufallszahlen erscheinen auf dem Bildschirm.

```
RANDOMIZE 3
x=RND
RANDOMIZE 3
PRINT x,RND
```

--> Es erscheint zweimal die gleiche "Zufallszahl" auf dem Monitor.

Integer-Arithmetik

Befehle und Funktionen

DEC, INC
ADD, SUB, MUL, DIV
PRED(), SUCC()
ADD(), SUB(), MUL(), DIV(), MOD()

DEC, INC
ADD, SUB, MUL, DIV

Diese Befehle sind eine verkürzte Schreibweise für folgende häufige Ausdrücke:

```
DEC x     entspricht  x=x-1
INC x     entspricht  x=x+1
ADD x,y   entspricht  x=x+y

SUB x,y   entspricht  x=x-y
MUL x,y   entspricht  x=x*y
DIV x,y   entspricht  x=x/y
```

Die links stehenden Befehle benötigen für ihre Ausführung deutlich weniger Zeit als die rechts stehenden Ausdrücke. Dieser Unterschied ist bei Integervariablen besonders groß.

Wichtig ist, daß INC, DEC, ADD, SUB, MUL und DIV bei Integervariablen (%,&,|) keine Überlaufprüfung durchführen.

DEC i
INC i

```
i: avar
```

Die Befehle DEC und INC dienen der Veränderung eines Wertes um eins. DEC vermindert (dekrementiert) den Wert der numerischen Variable i um eins, während INC den Wert von i um eins erhöht (inkrementiert).

Diese Befehle arbeiten auch mit Fließkomma-Variablen, sind jedoch mit Integer-Variablen erheblich schneller.

Beispiele:

```
x%=4
y%=7
DEC x%
INC y%
PRINT x%,y%
```

--> Auf dem Bildschirm erscheinen die Zahlen 3 und 8.

```
a|=255
INC a|
INC a|
PRINT a|
```

--> Ergibt 1 !!, da keine Fehlerprüfung.

ADD x,y
SUB x,y
MUL x,y
DIV x,y

```
x: avar
y: aexp
```

Der Befehl ADD erhöht die Variable x um den Wert y, während SUB x um den Wert von y vermindert. MUL multipliziert x mit y und weist das Ergebnis x zu. DIV teilt die Variable x durch y und legt das Resultat in x ab.

Bei diesen Befehlen muß x eine numerische Variable sein, während y ein numerischer Ausdruck ist.

Diese Befehle arbeiten auch mit Fließkomma-Variablen, sind jedoch mit Integer-Variablen erheblich schneller.

Beispiel:

```
x%=1
y%=2
z%=3
ADD x%,y%          !nun ist x gleich 3
SUB z%,(x%-1)/2    !der numerische Ausdruck (x%-1)/2 ergibt 1
PRINT x%,y%
```

--> Auf dem Bildschirm erscheinen die Zahlen 3 und 2.

PRED(), SUCC()
ADD(), SUB(), MUL(), DIV(), MOD()

PRED und SUCC ermitteln die nächstgrößere oder nächstkleinere Zahl. ADD(), SUB(), MUL(), DIV() und MOD() bieten eine schnelle Integerarithmetik mit Polnischer Notation.

PRED(i)
SUCC(i)

i: iexp

PRED gibt die nächstkleinere und SUCC die nächstgrößere Zahl zurück. PRED und SUCC liefern also den Vorgänger (predecessor) bzw. den Nachfolger (successor) eines numerischen Ausdrucks. Beide Funktionen arbeiten mit Integerarithmetik, Nachkommastellen werden also ignoriert.

Diese beiden Funktionen stehen auch für String-Ausdrücke zur Verfügung (siehe auch im Abschnitt über Zeichenkettenverwaltung).

Beispiel:

```
i%=6
j%=PRED(i%)
PRINT j%,SUCC(2),PRED(3*i%)
```

--> Auf dem Bildschirm erscheinen die Zahlen 5, 3 und 17.

ADD(x,y)
SUB(x,y)
MUL(x,y)
DIV(x,y)
MOD(x,y)

```
x,y: iexp
```

Diese Funktionen können an Stelle folgender numerischer Operatoren verwendet werden:

```
ADD(x,y)  entspricht  x+y
SUB(x,y)  entspricht  x-y
MUL(x,y)  entspricht  x*y
DIV(x,y)  entspricht  x\y        x DIV y
MOD(x,y)  entspricht  x MOD y
```

Da die o.g. Funktionen mit Integerarithmetik arbeiten, werden Nachkommastellen ignoriert. Nach den Anweisungen

```
x%=5
y%=4
ADD y%,3
z%=SUB(x%,3)
```

hat x% den Wert 5, y% den Wert 7 und z% den Wert 2.

Die Funktionen ADD, SUB, MUL, DIV und MOD können beliebig verschachtelt werden. Diese Reihenfolge von Operatoren und Operanden nennt man Polnische Notation.

Beispiele:

```
DEFINT "a-z"
x=4
y=ADD(x,x)    !y wird gleich 8
z=SUB(x,2)
PRINT y,z,ADD(x,MUL(y,2))
```

--> Auf dem Bildschirm erscheinen die Zahlen 8, 2 und 20.

```
DEFINT "a-z"
x=2
y=MUL(x,3)                    !y wird gleich 6
PRINT y,DIV(8,x),MOD(11,4)    !MOD(11,4) ist gleich 3
```

--> Auf dem Bildschirm erscheinen die Zahlen 6, 4 und 3.

```
DEFINT "a-z"
x=5
y=ADD(SUB(x,2),MUL(3,4))
PRINT y,DIV(8,MOD(14,4))
```

--> Auf dem Bildschirm erscheinen die Zahlen 15 und 4.

Bit-Operationen

BCLR, BSET, BTST, BCHG
SHL, SHR, ROL, ROR
AND(), OR(), XOR(), IMP(), EQV()
SWAP()
BYTE(), CARD(), WORD()

Die Bit-Operationen beeinflussen numerische Ausdrücke auf Bit-Ebene. Die (Assembler-Programmierern vertrauten) Kommandos BCLR, BSET, BTST und BCHG löschen, setzen, testen und negieren Bits. SHL, SHR, ROL und ROR verschieben oder rotieren Bits.

Die Funktionen AND, OR, XOR, IMP und EQV sind logische Verknüpfungen. Dabei gilt folgende Zählweise: 0 ist das niederstwertige Bit; bei 4-Byte-Integerwerten ist 31 das höchstwertige Bit und zugleich das Vorzeichen-Bit (ist das Vorzeichen-Bit gesetzt, wird die negative Zahl im 2er-Komplement dargestellt, sonst ist es eine positive Zahl).

SWAP vertauscht die beiden Wörter eines 4-Byte-Werts. BYTE liest die unteren 8 Bits und CARD die unteren 16 Bits eines Integer-Ausdrucks. WORD erweitert ein Wort auf Langwort, d.h. Bit 15 wird in Bit 16 bis Bit 31 kopiert.

BCLR(x,y)
BSET(x,y)
BCHG(x,y)
BTST(x,y)

```
x,y: iexp
```

Diese Befehle erlauben das Löschen, Setzen, Negieren und Testen eines Bits. Dabei beginnt die Zählung der Bits bei Null. Die Bit-Nummer liegt im Bereich 0 bis 31 und wird Prozessor-intern durch AND 31 maskiert.

Die Funktion BCLR setzt das y-te Bit des numerischen Ausdrucks x gleich null. BSET setzt analog dazu Bit-Nummer y von x gleich 1. BCHG setzt Bit y von x gleich 1, wenn es vorher 0 war und gleich 0, wenn es vorher 1 war. Die Funktion BTST ergibt -1 (TRUE), wenn Bit y von x gleich 1 ist und 0 (FALSE), wenn dieses Bit gleich 0 ist.

Beispiele:

```
x=BSET(0,3)
PRINT x,BSET(0,5)
```

--> Es erscheinen die Zahlen 8 (2^3) und 32 (2^5) auf dem Monitor.

```
REPEAT
  t|=Inp(2)
  PRINT CHR$(t|),CHR$(BCLR(t|,5))
UNTIL CHR$(t|)="x"
```

--> Beim Drücken einer Buchstabentaste erscheint der Buchstabe und der dazugehörende groß geschriebene Buchstabe (bei Kleinbuchstaben ist Bit 5 immer gesetzt, das Löschen dieses Bits erzwingt die Umwandlung in einen Großbuchstaben).

```
s$="Testwort"
FOR i%=1 TO LEN(s$)
  PRINT CHR$(BCHG(ASC(MID$(s$,i%)),5));
NEXT i%
```

--> Gibt 'tESTWORT' auf dem Bildschirm aus, verwandelt also jeden Klein- in einen Großbuchstaben und umgekehrt. Dies gilt jedoch nicht für Umlaute.

SHL(x,y) **SHL&**(x,y) **SHL|**(x,y)
SHR(x,y) **SHR&**(x,y) **SHR|**(x,y)
ROL(x,y) **ROL&**(x,y) **ROL|**(x,y)
ROR(x,y) **ROR&**(x,y) **ROR|**(x,y)

x,y: iexp

Diese Befehle verschieben (SHift) oder rotieren (ROtate) den Inhalt eines numerischen Ausdrucks x um y Bit. Ohne Angabe eines speziellen Variablentyps geschieht dies auf Langwortlänge (4 Bytes), bei Angabe eines '&' auf Wortlänge (2 Bytes) und bei Angabe eines '|' auf Bytelänge. Der dritte Buchstabe des Funktionsnamens gibt die Richtung der Verschiebung oder Rotation an. Dabei steht 'L' für links (Left) und 'R' für rechts (Right).

Bei Word-Funktionen(&) wird das Bit 15 anschließend in die Bits 16 bis 31 kopiert, und bei Byte-Funktionen(|) werden die Bits 8 bis 31 gelöscht.

Die folgenden Zahlenbeispiele zeigen die Wirkung der Shift-Befehle.

```
x%      SHL|(x,1)      BIN$(x%,16)          BIN$(SHL|(x%,1),16)
18      36             00000000 00010010    00000000 00100100
642     4              00000010 10000010    00000000 00000100

x%      SHL&(x%,1)     BIN$(x%,16)          BIN$(SHL&(x%,1),16)
18      36             00000000 00010010    00000000 00100100
130     4              00000000 10000010    00000001 00000100

x%      SHR&(x%,2)     BIN$(x%,16)          BIN$(SHR&(x%,2),16)
24      6              00000000 00011000    00000000 00000110
4162    1040           00010000 01000010    00000100 00010000
```

(Die Bits sind nur aus Gründen der Übersichtlichkeit in Achter-Gruppen zusammengefaßt worden, BIN$ gruppiert nicht.)

Die nächsten Zahlenbeispiele betreffen die Rotier-Befehle. Bei diesen werden Bits, die den betroffenen Zahlenbereich verlassen, auf der gegenüberliegenden Seite wieder hineingeschoben.

Es sei z.B. nur das höchste Bit eines Bytes gesetzt. Dieses Byte wird nun um ein Bit nach links rotiert (ROL|(128,1)). Das links herausgeschobene Bit wird nun rechts wieder hereingeschoben, so daß das erste Bit des Funktionsergebnisses gesetzt ist. Bei SHL|(128,1) wäre das herausgeschobene gesetzte Bit gleich Null geworden.

Weitere Zahlenbeispiele sind:

```
x|      ROL|(x,1)      BIN$(x|,8)     BIN$(ROL|(x|,1),8)
6       12             00000110       00001100
130     5              10000010       00000101

x|      ROR|(x|,3)     BIN$(x|,8)     BIN$(ROR|(x|,3),8)
66      144            01000010       00000000
2       64             00000010       01000000
```

Beispiel:

```
x|=128+1                    !setzt Bit 7 und 0
x|=ROR|(x|,1)               !y wird gleich 192
PRINT SHL(y%,4),y%*2^4
PRINT SHL(ROR|(128+1,1),4)
```

--> Es erscheint dreimal die Zahl 3072 auf dem Monitor. Dem Ausdruck a*2^b entspricht die Funktion SHL(a,b), sofern kein Bit über den vier Byte langen Geltungsbereich geschoben wird. Die Formulierung, die die Bit-Funktion benutzt, ist allerdings deutlich schneller.

AND(x,y)
OR(x,y)
XOR(x,y)
IMP(x,y)
EQV(x,y)

```
x,y: iexp
```

Diese Funktionen sind logische Verknüpfungen zweier numerischer Ausdrücke. Im Funktionsergebnis von AND sind nur Bits gesetzt, die sowohl in x als auch in y gesetzt sind. Das Funktionsergebnis von OR enthält gesetzte Bits an den Stellen, an denen Bits in x oder y oder in beiden Ausdrücken gesetzt sind. XOR setzt nur Bits, die in x oder y, aber nicht in x und y gleichzeitig gesetzt sind. Anders formuliert: XOR setzt Bits, die in x und y verschiedene Werte haben.

IMP weist Bits nur dann eine Null zu, wenn das entsprechende Bit in x gesetzt und in y nicht gesetzt ist, ansonsten wird das Bit im Funktionsergebnis gesetzt. EQV setzt ein Bit des Funktionsergebnisses, wenn die entsprechenden Bits in x und y dieselben Werte haben. Betrachten Sie zur Erläuterung dieser Kommandos auch die Wahrheitstabellen im Abschnitt über logische Operatoren.

Beispiel:

```
x=3
y=2
z=AND(x,y)  !z wird gleich 2
PRINT OR(2,7),XOR(x,1+4+8)
```

--> Auf dem Bildschirm erscheinen die Zahlen 7 und 14.

```
PRINT BIN$(15,4),15
PRINT BIN$(6,4),6
PRINT BIN$(IMP(15,6),4),"IMP(15,6)"
PRINT BIN$(EQV(15,6),4),"EQV(15,6)"
```

--> Auf dem Bildschirm erscheint:

```
1100      15
0101      6
0111      IMP(15,6)
0110      EQV(15,6)
```

SWAP(x)

```
x: iexp
```

Die Funktion SWAP faßt den numerischen Ausdruck x als Langwort (4 Byte) auf und vertauscht dessen oberes und unteres Wort (jeweils 2 Byte). Diese Funktion hat nichts mit dem gleichnamigen GFA-BASIC-Befehl zu tun und wird für verschiedene Zwecke gebraucht (z.B. um einen Langwortparameter in zwei Worten an eine Betriebssystemroutine zu übergeben oder um Pointer mit vertauschter Wortreihenfolge zu bearbeiten).

Beispiel:

```
x=1044480
PRINT BIN$(x,32)
y=SWAP(x)
PRINT BIN$(y,32)
```

--> Auf dem Monitor erscheint:

```
00000000000011111111000000000000
11110000000000000000000000001111
```

Ein Anwendungsbeispiel ist:

```
~WIND_SET(0,13,SWAP(t%),t%,0,0)
```

BYTE(x)
CARD(x)
WORD(x)

x: iexp

BYTE gibt die unteren 8 Bits des numerischen Ausdrucks x zurück. Entsprechend liest CARD die unteren 16 Bits von x. WORD erweitert ein Wort auf Langwortlänge (d.h. Bit 15 wird in Bit 16 bis Bit 31 kopiert).

Beispiel:

```
PRINT BYTE(1+254),BYTE(1+255)
PRINT HEX$(CARD(&H1234ABCD))
```

--> Schreibt 255, 0 und ABCD auf den Bildschirm.

5 - Zeichenkettenverwaltung

LEFT$, RIGHT$
MID$ (als Funktion)
PRED, SUCC
LEN
INSTR, RINSTR
STRING$, SPACE$, SPC
UPPER$
LSET, RSET, MID$ (als Befehl)

Die Befehle LEFT$ und RIGHT$ liefern den linken oder rechten Teil einer Zeichenkette zurück. MID$ kann, wenn es als Funktion verwendet wird, in der Mitte stehende Teile eines Strings zurückmelden. Wird MID$ als Befehl verwendet, so kann man mit ihm einen String in einen anderen String einsetzen. PRED und SUCC liefern das Zeichen mit dem um eins niedrigeren bzw. höheren ASCII-Wert als des angegebenen String-Ausdrucks.

LEN bestimmt die Länge einer Zeichenkette, INSTR und RINSTR suchen in einem String nach einer anderen Zeichenkette. STRING$, SPACE$ und SPC dienen der Erzeugung von Zeichenketten, die denselben String-Ausdruck mehrmals enthalten. UPPER$ verwandelt alle Zeichen eines Strings in Großbuchstaben. Das links- und rechtsbündige Einsetzen eines Strings in einen anderen erfolgt mit LSET und RSET.

**LEFT$**(a$ [,x])
**RIGHT$**(a$ [,x])

```
a$: sexp
x: iexp
```

LEFT$ liefert die ersten x-Zeichen der Zeichenkette a$. Ist x größer als die Anzahl der Zeichen in a$, so wird a$ komplett übergeben. Ohne die Angabe von x wird nur das erste Zeichen des Strings a$ gemeldet. RIGHT$ ergibt analog dazu die letzten x-Zeichen von a$. Ohne Angabe von x wird das letzte Zeichen von a$ bestimmt.

Beispiele:

```
a$="Handbuch für GFA-BASIC"
b$=LEFT$("GFA-Systemtechnik",4)
PRINT b$;RIGHT$(a$,5)
```

--> Es wird das Wort 'GFA-BASIC' auf den Monitor geschrieben.

```
a$="Oberhausen"
b$=LEFT$(a$)+RIGHT$("Technik")
PRINT b$
```

--> Es erscheint 'Ok' auf dem Bildschirm.

MID$**(a$,x [,y]) **(als Funktion)

```
a$: sexp
x,y: iexp
```

Die Funktion MID$ gibt y-Zeichen ab der Position x des Strings a$ zurück. Ist x größer als die Länge von a$, so wird eine leere Zeichenkette zurückgegeben. Läßt man y weg, so wird der gesamte Teil-String ab dem x-ten Zeichen ermittelt.

Beispiel:

```
a$="Handbuch für GFA-BASIC"
b$=MID$(a$,14,9)+MID$("Version 3.0",8)
PRINT b$
```

--> Es erscheint der Text 'GFA-BASIC 3.0' auf dem Bildschirm.

PRED(a$)
SUCC(a$)

a$: sexp

PRED liefert das Zeichen mit dem um eins verringerten ASCII-Wert des zu übergebenden String-Ausdrucks. Dabei wird jeweils nur das erste Zeichen einer Zeichenkette ausgewertet. Analog liefert SUCC das um eins erhöhte Zeichen. Dabei entspricht PRED(a$) dem Ausdruck CHR$(PRED(ASC(a$))) und SUCC(a$) CHR$(SUCC(ASC(a$))).

Diese beiden Funktionen stehen auch für Integerausdrücke zur Verfügung (siehe Abschnitt über Integer-Arithmetik).

Beispiel:

```
zeichen$="B"
vorgaenger$=PRED(zeichen$)
nachfolger$=SUCC(zeichen$)
PRINT vorgaenger$,zeichen$,nachfolger$
```

--> Gibt die Buchstaben A, B und C auf dem Bildschirm aus.

LEN(a$)
**TRIM$**(a$)

```
a$: sexp
```

LEN ermittelt, wie viele Zeichen in a$ enthalten sind und meldet diese Zahl zurück.

TRIM$ entfernt Leerzeichen am linken und rechten Rand eines Strings.

Beispiele:

```
a$="Test"
x=LEN(a$)+1
PRINT x,LEN("Wort")
```

--> Es erscheinen die Zahlen 5 und 4 auf dem Bildschirm.

```
b$="   test   "
PRINT LEN(b$)
PRINT TRIM$(b$)
PRINT LEN(TRIM$(b$))
```

--> Ausgegeben wird 10, der String "test" ohne Leerzeichen und 4.

INSTR(a$,b$)
INSTR(a$,b$, [x])
INSTR([x] ,a$,b$)

```
a$,b$: sexp
x: iexp
```

Die Funktion INSTR sucht in der Zeichenkette a$ nach dem String b$. Ist x angegeben, so wird in a$ ab dem x-ten Zeichen mit der Suche begonnen, ansonsten beginnt die Suche beim ersten Zeichen. Es wird die Position zurückgemeldet, ab der b$ in a$ enthalten ist. Ist b$ nicht gefunden worden, so ist INSTR gleich Null. Wenn a$ und b$ Leer-Strings sind, ist INSTR gleich Eins.

Beispiel:

```
a$="GFA-Systemtechnik"
x=INSTR(a$,"System")
PRINT x,INSTR("GFA-BASIC","BASIC",6)
```

--> Es erscheinen die Zahlen 5 und 0 auf dem Bildschirm.

RINSTR(a$,b$)
RINSTR(a$,b$, [x])
RINSTR([x] ,a$,b$)

```
a$,b$: sexp
x: iexp
```

RINSTR() durchsucht wie INSTR einen String, beginnt allerdings die Suche am Ende der Zeichenkette.

Beispiel:

```
PRINT RINSTR("A:\ORDNER\*.GFA","\")
```

--> Sucht nach dem letzten Rückwärtsschrägstrich '\' im Pfadnamen und gibt die Position des gesuchten Zeichens (in diesem Fall 10) aus.

**STRING$**(x,a$)
STRING$(x,code)
SPACE$(x)
SPC(x)

```
x,code: iexp
a$: sexp
```

Die Funktion STRING$ erzeugt einen String, der den String-Ausdruck 'a$' oder den ASCII-Wert 'code' insgesamt x-mal (zwischen 0 und 32767) enthält. Bei Angabe von 'code' wird CHR$(code) verwendet.

SPACE$ erzeugt einen String, der x Leerzeichen enthält. SPC gibt in einem PRINT-Befehl x Leerzeichen aus. SPC kann im Unterschied zu SPACE$ keinen String erzeugen, den man z.B. über ein Gleichheitszeichen einer Variablen zuweisen könnte.

Beispiel:

```
a$="GFA "
b$=SPACE$(5)
PRINT b$;STRING$(3,a$);SPC(4);STRING$(5,"<")
```

--> Auf dem Bildschirm erscheint ' GFA GFA GFA < < < < <'.

**UPPER$**(a$)

```
a$: sexp
```

Wandelt alle kleingeschriebenen Buchstaben einer Zeichenkette in Großbuchstaben um. Dies funktioniert auch mit Umlauten.

Beispiel:

```
a$="Test"
b$=UPPER$(a$)+UPPER$("Wo"+"rt")
PRINT b$;UPPER$(" Gfa-Basic 3.0")
```

--> Auf dem Bildschirm erscheint 'TESTWORT GFA-BASIC 3.0'.

LSET a$ = b$
RSET a$ = b$
**MID$**(a$,x [,y]) (als Befehl)

```
a$: svar
b$: sexp
x,y: iexp
```

LSET und RSET setzen den String-Ausdruck b$ links- (LSET) bzw. rechtsbündig in die String-Variable a$ ein und füllen gegebenenfalls mit Leerzeichen auf.

MID$ als Befehl ermöglicht das Einsetzen eines String-Ausdrucks in die Mitte einer String-Variable. So würde bei

```
MID$(a$,x,y)=b$
```

in die Variable a$ ab dem x-ten Zeichen der String-Ausdruck b$ eingesetzt. Der optionale Parameter y legt fest, wie viele Zeichen maximal von b$ in a$ eingesetzt werden. Wird y weggelassen, so werden so viele Zeichen wie möglich in a$ übernommen. Durch MID$ wird die Länge von a$ nicht verändert, sondern die aus b$ kommenden Zeichen überschreiben Teile von a$. Ist a$ zu kurz, um b$ ganz aufzunehmen, so wird das Einsetzen abgebrochen.

Beispiele:

```
a$="      "
FOR i%=1 TO 128          ! gibt linksbündig formatierte
  LSET a$=STR$(i%)
  PRINT a$;
NEXT i%
PRINT
FOR i%=1 TO 128
  RSET a$=STR$(i%)       ! und rechtsbündig formatierte
  PRINT a$;              ! Zahlenkolonnen aus
NEXT i%
~INP(2)
```

--> Gibt linksbündig und rechtsbündig formatierte Zahlenkolonnen aus. Statt RSET a$ = STR$(i%) kann auch a$ = STR$(i%,5,0) verwendet werden.

```
a$="GF   ASIC"
MID$(a$,3)="A-B"
b$="Testwort"
MID$(b$,5,2)="wer war das ?"
PRINT a$,b$
```

--> Es erscheinen die Worte 'GFA-BASIC' und 'Testwert'.

6

6 - Tastaturein- und Bildschirmausgabe

In diesem Kapitel werden grundlegende Ein- und Ausgabemöglichkeiten vorgestellt. Es beginnt mit der Abfrage eines Zeichens von der Tastatur mit INKEY$. Danach folgt die Abfrage einer Variablen mittels INPUT und der verwandten Befehle LINE INPUT, FORM INPUT und FORM INPUT AS.

Die Besprechung der Ausgabemöglichkeiten beginnt mit dem einfachsten Befehl, mit PRINT. Anschließend werden dessen erweiterte Versionen besprochen (PRINT AT, PRINT USING). Danach werden die Befehle zur Abfrage (CRSCOL, CRSLIN, POS) zur Bestimmung der Cursor-Position (TAB, HTAB, VTAB) besprochen.

Am Ende dieses Abschnittes werden die KEYxxx-Befehle vorgestellt. Mit Hilfe dieser neuen Anweisungen ist eine einfache Handhabung von tastaturabhängigen Funktionen während des Programmablaufs möglich.

INKEY$

INKEY$ liest ein Zeichen von der Tastatur. Diese Funktion ist allerdings nicht in der Lage, die Tastaturumschalttasten abzufragen (Shift, Alternate, Control, CapsLock). INKEY$ wartet nicht auf einen Tastendruck, sondern meldet einen Leer-String zurück, wenn seit der letzten Tastaturabfrage keine Taste mehr gedrückt wurde. Ansonsten meldet die Funktion das ASCII-Zeichen der gedrückten Taste zurück. Hat die gedrückte Taste aber keinen ASCII-Code, wie z.B. die Funktionstasten oder die Help- und Undo-Taste, so wird der Scan-Code der gedrückten Taste ermittelt. Dabei wird ein zwei Zeichen langer String zurückgemeldet, der im ersten Zeichen den Code CHR$(0) und im zweiten die entsprechende Kennung der Sondertaste enthält. Folgendes Beispiel gestattet die Ermittlung der mit Inkey$ zu erhaltenden Werte.

Beispiel:

```
DO
  t$=INKEY$
  IF t$<>""
    IF LEN(t$)=1
      PRINT "ASCII-Code: ";ASC(t$),"ASCII-Zeichen: ";t$
    ELSE
      PRINT "CHR$(0), Scan-Code: ";CVI(t$)
    ENDIF
  ENDIF
LOOP
```

--> Zeigt bei jeder gedrückten Taste ihren ASCII- oder Scan-Code an. Anmerkung: CVI(t$) entspricht hier ASC(RIGHT$(t$)), da ASC(t$) = 0.

INPUT ["text",] x [,y,...]
INPUT ["text";] x [,y,...]

`x,y: avar oder svar`

Der Befehl INPUT kann in mehreren Variationen verwendet werden. Er dient der Eingabe von Variablen oder Variablenlisten mit oder ohne vorangestellte Erläuterungen ("text"). INPUT bezieht sich immer auf die letzte Cursor-Position. Mit Hilfe von PRINT AT, gefolgt von einem Semikolon oder LOCATE, VTAB, HTAB, kann der Cursor plaziert werden, um die Position der Eingabe festzulegen.

Wenn dem Befehlswort INPUT ein Text folgt, so kann dieser von den nachfolgenden Variablen durch ein Komma oder ein Semikolon getrennt werden. Wird ein Semikolon verwendet, so werden ein Fragezeichen und ein Leerzeichen an die Beschreibung angehängt und der Cursor dahinter plaziert. Verwendet man ein Komma, wird der Cursor für die Eingabe direkt hinter dem letzten Zeichen der Beschreibung plaziert.

Wenn kein Text hinter INPUT steht, so erscheint in jedem Fall ein Fragezeichen, gefolgt von einem Leerzeichen, hinter dem der Cursor steht.

Wird nur eine einzige Variable abgefragt, so muß deren Eingabe durch das Drücken der Return- oder Enter-Taste bestätigt werden. Wenn mehrere Variablen mit einem INPUT-Befehl erfragt werden, so kann jede einzelne Variable durch Drücken von Return oder Enter bestätigt werden, die Variablen können aber auch durch Kommas getrennt und mit einem einzigen Druck der Return- oder Enter-Taste eingegeben werden. Wenn man dennoch Kommas in einem String per INPUT eingeben will, muß man den Befehl LINE INPUT verwenden.

Wurde eine numerische Variable erwartet, aber ein Text eingegeben, so ertönt ein Fehlersignal, und die Eingabe muß erneut begonnen werden. Bis zum Drücken der Return- oder Enter-Taste können mit Hilfe von Backspace- und Delete-Taste Zeichen innerhalb der Eingabe gelöscht werden; auch die linke und rechte Cursor-Taste sind aktiv. Durch Drücken

der Insert-Taste kann zwischen Einfüge- und Überschreib-Modus hin- und hergeschaltet werden. Die maximale Eingabelänge beträgt 255 Zeichen.

Sonderzeichen können auf drei verschiedene Arten eingegeben werden:

- Durch Drücken von <Alternate> und den Ziffern auf dem Ziffernblock, z.B.: Bei gehaltener Alternate-Taste die Zahl 64 auf dem Zehnerblock drücken. Bei Loslassen der Alternate-Taste wird @ erscheinen. Dies gilt auch für INKEY$, INP(2), GEM-Dialoge usw., sofern es nicht durch KEYPAD abgeschaltet wird.
- Durch Eingabe von <Control><S> und einem weiteren Zeichen, z.B.: <Control><S><c> für das Pi-Zeichen.
- Durch Eingabe von <Control><A> und dem nachfolgenden ASCII-Code des gewünschten Zeichens, abhängig von KEYPAD, vgl. dort z.B.: <Control><A><2><7> für das Escape-Zeichen.

Beispiel:

```
INPUT a$
INPUT "",b$
INPUT "Zwei Zahlen bitte: ";x,y
PRINT a$,b$,x,y
```

--> Liest zwei Strings und zwei numerische Variablen ein. Der erste INPUT-Befehl meldet sich mit '? ', der zweite ohne Text und der dritte mit der Aufforderung 'Zwei Zahlen bitte: ?'

LINE INPUT ["text",] a$ [,b$...]
LINE INPUT ["text";] a$ [,b$...]

`a$,b$: svar`

LINE INPUT ist eine Variante des INPUT-Befehls. Im Unterschied zu INPUT ist es hier möglich, Kommas in einer String-Variablen einzugeben. Vorangehende Beschreibungen, Eingabe von Variablen oder einer Variablenliste, die Korrektur der Eingabe bis zum Drücken der Return- oder Enter-Taste und weitere Variationsmöglichkeiten werden wie bei INPUT gehandhabt, es ist allerdings nur für String-Variablen möglich.

Beispiel:

```
LINE INPUT a$
INPUT b$
PRINT a$,b$
```

--> Geben Sie bitte zweimal den Text 'Kom,ma' ein. Es erscheint danach 'Kom,ma' und 'Kom' auf dem Monitor; Vgl. auch LINE INPUT #.

FORM INPUT n,a$
FORM INPUT n **AS** a$

```
n: iexp
a$: svar
```

FORM INPUT und FORM INPUT AS dienen beide der Eingabe von String-Variablen. Dabei gibt n an, wie viele Zeichen maximal im String a$ sein dürfen (zwischen 1 und 255).

FORM INPUT AS gibt zusätzlich den aktuellen Wert von a$ aus, der dann editiert werden kann. Die Editiermöglichkeiten sind bei beiden Befehlen dieselben wie beim INPUT-Befehl.

Beispiel:

```
FORM INPUT 10,a$
b$="test"
FORM INPUT 5 AS b$
PRINT a$,b$
```

--> Fragt zwei Strings ab. Bei der Abfrage des zweiten Strings wird das Wort 'test' als voreingestellter Wert von b$ zur Editierung freigegeben.

PRINT
PRINT ausdruck
PRINT AT(spalte,zeile);ausdruck
WRITE ausdruck
LOCATE zeile,spalte

```
ausdruck: beliebige sexp oder aexp oder deren Kombination
spalte,zeile: iexp
```

Der Befehl PRINT ohne folgende Parameter bewirkt einen Zeilenvorschub. Befindet sich der Cursor bereits in der letzten Zeile, wird der gesamte Bildschirm um eine Zeile nach oben verschoben. PRINT, von einem Ausdruck gefolgt, bewirkt die Ausgabe dieses Ausdrucks an der aktuellen Cursor-Position. Zeichenketten müssen in Anführungszeichen eingeschlossen werden. Besteht der auszugebende Ausdruck aus mehreren Elementen (Konstanten, Variablen oder Ausdrücken), so können die einzelnen Teile durch Semikolon, Komma oder Apostroph getrennt werden.

Die Verwendung des Kommas bewirkt die Plazierung des Cursors hinter die nächste durch 16 teilbare Spaltenposition. Ist die letzte Spalte erreicht, wird der Cursor in die Spalte 17 der nächsten Zeile bewegt. Das Semikolon bewirkt die Ausgabe der betreffenden Elemente ohne Zwischenraum. Verwendet man das Apostroph, wird ein Leerzeichen zwischen den entsprechenden Elementen eingefügt.

PRINT AT ermöglicht die Positionierung des auszugebenden Ausdrucks ab einer bestimmten Spalte und Zeile. Dabei stehen je nach Auflösung bis zu 80 Spalten und 25 Zeilen zur Verfügung. Die Verwendung von Fenstern schränkt den Wertebereich für die Plazierung des Cursors ebenfalls ein.

Wenn der Ausgabeausdruck nicht mit einem Semikolon beendet wird, so wird der Cursor an den Anfang der nächsten Zeile plaziert. Falls er schon in der letzten Zeile stand, so wird der Bildschirm um eine Zeile nach oben geschoben.

Wenn mit PRINT Control-Zeichen (Codes bis 31) angegeben werden, so werden diese durch den VT-52-Emulator verarbeitet (siehe Anhang).

Im Gegensatz zu PRINT AT setzt LOCATE lediglich den Cursor an die angegebene Zeilen- und Spaltenposition. Es ist nicht möglich, Ausdrücke mit LOCATE auszugeben (Vgl. VTAB/HTAB).

Der WRITE-Befehle dient der Datenspeicherung in sequentiellen Dateien zum Einlesen mit Input. Nach dem WRITE-Befehl stehen numerische und String-Ausdrücke, die durch Kommata voneinander abgetrennt sind.

Bei der Ausgabe werden die Ausdrücke durch Kommata voneinander abgetrennt; String-Ausdrücke werden in Anführungszeichen eingeschlossen. Anmerkung: Dieses Format eignet sich bei Ausgabe auf Diskette besonders für das Wiedereinlesen mit INPUT. Hinter dem letzten Ausdruck eines WRITE-Befehls kann auch ein Semikolon stehen, dann wird die abschließende Ausgabe von CR/LF unterdrückt.

Beispiele:

```
a$="GFA-Systemtechnik"
PRINT Left$(a$,4)+"BASIC"'1+2;
PRINT ".0","GFA-";UPPER$(MID$(a$,5))
```

--> Auf dem Bildschirm erscheint der Text 'GFA-BASIC 3.0' und 'GFA-SYSTEMTECHNIK'.

```
PRINT AT(4,8);"in der vierten Spalte der achten Zeile"
```

--> Schreibt einen String an die Position 4, 8.

```
LOCATE 8,4
PRINT "in der achten Zeile, an der vierten Spalte"
```

--> Positioniert den Cursor an die vierte Spalte der achten Zeile und gibt mit dem darauffolgenden PRINT an dieser Stelle einen String aus.

```
WRITE 1+1,"Hallo",3*4
```

--> Ergibt die Ausgabe 2,"Hallo",12

PRINT USING format$,ausdruck [;]
PRINT AT(zeile,spalte);**USING** format$,ausdruck [;]

```
format$: sexp
ausdruck: beliebig viele durch Kommata getrennte sexp oder aexp
spalte,zeile: iexp
```

PRINT USING und seine Variation PRINT AT USING dienen der formatierten Datenausgabe am Bildschirm. Grundsätzlich arbeiten diese Befehle wie PRINT bzw. PRINT AT. Jedoch werden die im auszugebenden Ausdruck stehenden Daten gemäß dem Inhalt von format$ formatiert.

Zur Formatierung von numerischen Ausdrücken stehen folgende Zeichen zur Verfügung:

Platzhalter für eine Ziffer. Ist es die letzte Ziffer der Formatanweisung, so wird bei der Ausgabe aufgerundet.

. Dient zur Abtrennung der Dezimalziffern innerhalb mehrerer #-Zeichen.

, Fügt ein Komma an der jeweiligen Stelle zwischen den #-Zeichen ein und bewirkt so z.B. eine Trennung in den Tausender-Stellen.

- Das Minus-Zeichen darf nur an erster oder letzter Stelle des Format-Strings stehen. Es reserviert eine Stelle für die Ausgabe eines negativen Vorzeichens. Bei einem positiven Vorzeichen wird stattdessen ein Leerzeichen ausgegeben.

+ Analog zum Minus-Zeichen wird bei positiven Zahlen ein Plus-Zeichen vor- oder hintenangestellt. Plus- und Minus-Zeichen können nicht kombiniert werden.

* Ersatz für #, führende Nullen werden durch Sternchen ersetzt, statt Leerzeichen.

$ Bewirkt die Ausgabe eines $-Zeichens vor der Zahlenangabe, wenn es unmittelbar vor dem ersten #-Symbol steht.

^ Dient der Einstellung des Exponential-Formats (E+000). Dabei bestimmen die #-Zeichen die Länge der Mantisse. Die ^-Zeichen geben die Anzahl der Stellen für den Exponenten inklusive E+ bzw. E- an. Stehen mehrere # vor dem Dezimalpunkt, so wird der Exponent so angepaßt, daß er durch diese Anzahl teilbar ist. Für negative Zahlen muß unbedingt ein Vorzeichen vorgesehen werden.

Im Gegensatz zur Version 2.0 gilt jetzt der Grundsatz: Formatierung vor Richtigkeit. Das heißt, daß bei Überlauf nur die Ziffern in dem entsprechenden Bereich ausgegeben werden.

Für die Formatierung von Zeichenketten stehen folgende Zeichen zur Verfügung:

&	Bewirkt die Ausgabe der gesamten Zeichenkette.
!	Beschränkt die Ausgabe auf das erste Zeichen des Strings.
\...\	Gibt an, wie viele Zeichen (einschließlich der beiden \-Zeichen) einer Zeichenkette ausgegeben werden.
_	Der Unterstrich bewirkt die Ausgabe des folgenden Zeichens der Formatangabe.

Weiterhin ist die unveränderte Ausgabe von Texteinschüben zwischen diesen Formatangaben möglich. Die dem Format-String folgenden Variablen, Variablenlisten oder Ausdrücke werden durch Kommas getrennt.

Beispiel:

```
PRINT USING "#.####",PI
PRINT AT(4,4); USING "PI_._._. #.###",PI;
```

--> Es erscheinen die Texte '3.1416' und 'PI... 3.142'.

```
FOR i%=1 TO 14
  PRINT USING "###.##^^^^ ",2^i%;
NEXT i%
```

--> Ausgabe:

```
     1.00E+00   2.00E+00   4.00E+00   8.00E+00  16.00E+00
    32.00E+00  64.00E+00 128.00E+00 256.00E+00 512.00E+00
     1.02E+03   2.05E+03   4.10E+03   8.19E+03  16.38E+03
```

Mit MODE können Kommata und Punkte ausgetauscht werden.

MODE n

n: iexp

Durch MODE kann zwischen Dezimalpunkt und Tausender-Komma sowie Dezimalkomma und Tausender-Punkt gewählt werden. Außerdem wird mit MODE das Format der Datumsdarstellung gewählt. Punkt und Komma gelten für PRINT USING und STR$(x,v,n). Die Datumseinstellung gilt für DATE$, SETTIME, DATE$= und FILES.

Der Parameter n kann Werte zwischen 0 und 3 annehmen. Dafür ergeben sich dann die Einstellungen gemäß folgender Tabelle:

Parameter n	USING	DATE$
MODE 0	#,###.##	16.05.1988
MODE 1	#,###.##	05/16/1988
MODE 2	#.###,##	16.05.1988
MODE 3	#.###,##	05/16/1988

DEFNUM n

n: iexp

DEFNUM beeinflußt die Ausgabe von Zahlen durch den PRINT-Befehl und dessen Varianten. Alle Zahlenausgaben, die hinter einem DEFNUM-Befehl folgen, werden mit n Stellen durchgeführt (Vor- und Nachkommastellen, der Punkt wird nicht mitgerechnet). Die interne Rechengenauigkeit wird dadurch nicht verändert. Die n + 1'ste Stelle wird bei diesen Ausgaben zur Rundung herangezogen.

Beispiel:

```
PRINT 100/3
DEFNUM 5
PRINT 100/3
```

--> Läßt die Zahlen 33.333333333 und 33.333 auf dem Bildschirm erscheinen.

CRSCOL
CRSLIN
POS(x)
TAB(n)
HTAB spalte
VTAB zeile

```
n,spalte,zeile: iexp
x: aexp
```

Die Befehlsgruppe CRSCOL, CRSLIN, POS und TAB dient der Abfrage und Plazierung des Cursors. CRSCOL liefert die aktuelle Spaltenposition und CRSLIN die aktuelle Zeilenposition des Cursors. POS liefert die Zahl der seit dem letzten Wagenrücklauf (Carriage Return) am Bildschirm ausgegebenen Zeichen AND 255. Der Ausdruck x wird ignoriert.

Der von POS zurückgelieferte Wert muß jedoch nicht mit der aktuellen Cursor-Position übereinstimmen. Wird z.B. eine Zeichenkette mit 120 Zeichen ausgegeben, so steht der Cursor in der Spalte 40, POS liefert jedoch den Wert 120. Insbesondere bei Ausgabe von Steuerzeichen hat POS(0) wenig mit der Cursorspalte zu tun, da nur die Zeichen gezählt werden. Dabei wird ein LF (CHR$(10)) nicht beachtet, ein CR (CHR$(13)) setzt den Zähler auf Null zurück und ein BS (CHR$(8)) vermindert den Zähler um eins.

TAB(n) gibt soviel Leerzeichen aus, bis POS(0) den Wert von n erreicht. Ist n bereits überschritten, so erfolgt zunächst ein Zeilenvorschub (CR/LF) wie bei PRINT ohne Parameter. Wieder wird nur AND 255 verwendet.

Die Befehle HTAB und VTAB positionieren den Cursor an einer bestimmten Spalten- bzw. Zeilenposition.

Bei Cursor-Positionierungen hat die obere linke Ecke die Koordinate 1.

Beispiele:

```
PRINT AT(38,12);"Test ";
PRINT CRSCOL'CRSLIN
PRINT Tab(37);"Test ";
PRINT POS(0)
```

--> Auf dem Bildschirm erscheint in der Bildschirmmitte 'Test 43 12' und 'Test 42'.

```
PRINT AT(4,3);"Wort 1"
HTAB 4
VTAB 2
PRINT "Wort 2"
```

--> Gibt an der vierten Spalte der dritte Zeile den Text 'Wort 1' und an der vierten Spalte der zweiten Zeile den String 'Wort 2' aus.

Die KEYxxx-Anweisungen

Diese Gruppe von Anweisungen ermöglicht die Kontrolle über den Status der Tastaturumschalttasten während des Programmlaufs sowie das Belegen der Funktionstasten mit frei definierbaren Zeichenketten (maximal jeweils 31 Zeichen), die auch im GFA-BASIC-Editor zur Verfügung stehen.

KEYPAD n

```
n: iexp
```

Der numerische Ausdruck n wird bitweise ausgewertet und hat folgende Bedeutung:

Bit	Bedeutung	0	1
0	NUMLOCK	Aus	Ein
1	NUMLOCK	Nicht schaltbar	Schaltbar
2	CTRL-KEYPAD	Normal	Cursor
3	ALT-KEYPAD	Normal	ASCII-Eingabe
4	KEYDEF ohne ALT	Aus	Ein
5	KEYDEF mit ALT	Aus	Ein

Mit gesetztem Bit 1 ist der NUMLOCK-Modus eingeschaltet, d.h. der Ziffernblock ist wie bei einem 'PC' belegt (siehe auch im Abschnitt über den Editor).

Ist Bit 2 gesetzt, kann der Cursor durch Drücken von Control und einer Pfeiltaste gesteuert werden.

Bei gesetztem Bit 3 ist demnach während des Programmlaufs die Eingabe von ASCII-Zeichen durch Drücken der Alternate-Taste, gefolgt vom ASCII-Wert, möglich.

Bei gesetztem Bit 4 können F1 bis F10 und Shift-F10 bis Shift-F1 mit Strings belegt werden. Bei gesetztem Bit 5 gilt dies (nur), wenn zusätzlich die Alternate-Taste gedrückt wird.

Der ATARI ST hat normalerweise die Einstellung, die KEYPAD 0 entspricht. Durch den GFA-BASIC-Interpreter wird im Editor KEYPAD 46 voreingestellt (d.h. die Bits 1, 2, 3 und 5 sind gesetzt).

KEYTEST n
KEYGET n
KEYLOOK n

n: ivar

Die KEYTEST-Funktion entspricht INKEY$ und liest ein Zeichen von der Tastatur, wenn seit der letzten Eingabe eine Taste (außer Alternate, Control, Shift und Caps-Lock) gedrückt wurde. Dabei wird Null geliefert, wenn keine Taste gedrückt wurde, sonst der ASCII-Wert des Zeichens (wie KEYGET).

KEYGET wartet auf einen Tastendruck. In n wird ein Langwort mit folgendem Aufbau zurückgemeldet: Bit 0-7: ASCII-Code, Bit 8-15: Null, Bit 16-23: Scan-Code, Bit 24-31: Status der Tastaturumschalttasten (kbshift).

KEYLOOK ermöglicht das Lesen aus dem Tastaturpuffer, ohne daß dessen Inhalt verändert oder gelöscht wird (Rückgabe wie KEYGET).

Bei Verwendung von KEYTEST, KEYGET und KEYLOOK erfolgt bei Angabe einer Byte-Variablen oder Wort-Variablen automatisch eine Konvertierung in einen 2-Byte-Integerwert (wie WORD(BIOS(2,2)).

Beispiele:

```
PRINT "Bitte <Esc> drücken"
REPEAT                    ! durchläuft die Schleife,
UNTIL INKEY$=CHR$(27)     ! bis die Esc-Taste gedrückt wird
'
PRINT "Bitte nochmal <Esc> drücken"
'
REPEAT                    ! macht das Gleiche,
  KEYTEST n|              ! aber anders
UNTIL n|=27
```

--> Wartet zweimal auf das Drücken der Escape-Taste.

```
PRINT "Bitte eine Taste drücken"
taste_1|=INP(2)                        ! auf Tastendruck warten
PRINT "Bitte noch eine Taste drücken"
KEYGET taste_2|                        ! das Gleiche noch einmal
PRINT "INP(2): ";taste_1|              ! und die ASCII-Werte
PRINT "KEYGET: ";taste_2|              ! der gedrückten Tasten ausgeben
```

--> Wartet zweimal auf einen Tastendruck und gibt die Codes der gedrückten Tasten aus.

```
REPEAT
  ' Viel rechnen
  KEYLOCK n%
UNTIL n%
LINE INPUT "Was willst Du?",a$
```

--> Wartet in einer - vorgetäuschten - Rechenschleife auf einen Tastendruck und liest dann einen String ein ohne die erste Taste zu unterschlagen.

```
DO
  KEYGET a%
  PRINT HEX$(a%,8)'BIN$(a%,32)'
  OUT 5,a%
  PRINT
LOOP
```

--> Wartet auf Tastendruck, gibt den zugehörigen Code in hex und binär aus und zeigt das Zeichen an. Anmerkung: Über OUT 5 erfolgt, wie auch bei VID:, ein Anzeigen von Steuerzeichen (<32).

KEYPRESS n

n: iexp

Der Befehl KEYPRESS simuliert das Drücken einer Taste. Dazu enthält der numerische Ausdruck n den ASCII-Wert der Taste, deren Drücken simuliert werden soll. Im Highword können zusätzlich der Scan-Code und der Status der Tastaturumschalttasten übergeben werden (vergleiche KEYGET).

Beispiele:

```
FOR i&=65 TO 90      ! simuliert das Drücken
  KEYPRESS i&        ! der Großbuchstaben
NEXT i&              ! A bis Z
'
KEYPRESS 13          ! simuliert das Drücken
'                    ! der Return-Taste
LINE INPUT a$        ! liest Zeichen bis zum ersten Wagenrücklauf
'                    ! d.h. CHR$(13) oder Return-Taste in die
PRINT a$             ! Zeichenkette a$ und gibt diese aus

KEYPRESS &H3B0000    ! drückt die Funktionstaste F1
KEYPRESS &1001B      ! drückt die Esc-Taste und zwar so, daß auch Eingaben
                     ! in GEM-gesteuerte Dialogboxen funktionieren. Das
                     ! GEM für seine Zwecke die Scan-Codes ab.
```

```
KEYDEF 1,"Hallo"+CHR$(13)
KEYPRESS &H83B0000
PAUSE 1
LINE INPUT a$
PRINT a$
```

--> Definiert einen Text, der bei Drücken von Alternate-F1 ausgegeben werden soll, simuliert dann das Drücken dieser Taste, wartet anschließend, damit die zugehörige Interrupt-Routine die Tasten-Definition abarbeiten kann und liest dann das "Hallo" per INPUT ein.

KEYDEF n,s$

```
n: iexp
s$: sexp
```

Der Befehl KEYDEF ermöglicht die Belegung der Funktionstasten mit beliebigen Zeichenketten, wobei diese Strings jeweils eine maximale Länge von 31 Zeichen haben dürfen. Dazu wird hinter dem Befehl KEYDEF ein arithmetischer Ausdruck zwischen 1 und 20 einschließlich angegeben. Mit n von 1 bis 10 können die Funktionstasten F1 bis F10 angesprochen werden. Für n von 11 bis 20 können Werte für die Shift-Taste und die Funktionstasten F1 bis F10 angegeben werden.

Die Belegung der Funktionstasten steht während des Programmablaufs und im Editor zur Verfügung. Im GFA-BASIC-Editor jedoch nur, wenn zusätzlich die Alternate-Taste gedrückt wird (vergleiche KEYPAD Bit 4 und 5).

Beispiel:

```
KEYDEF 1,"F1"
KEYDEF 11,"Shift+F1"
```

--> Gibt beim Drücken der Funktionstaste F1 den Text 'F1' und beim Drücken von Shift und F1 (jeweils in Verbindung mit Alternate) den Text 'Shift + F1' aus.

Allgemeine Ein- und Ausgabe

Im folgenden Kapitel wird zunächst das Einlesen von Konstanten (DATA, READ, RESTORE) erläutert. Anschließend geht es um die Verwaltung von Inhaltsverzeichnissen (DIR$, CHDIR, DIR, FILES, MKDIR, RMDIR). Im Zusammenhang mit diesen Befehlen wird der Aufbau des hierarchischen File-Systems erklärt.

Danach stehen das Öffnen, Schließen und Umbenennen von Dateien (EXIST, KILL, NAME, OPEN, CLOSE) im Vordergrund. Ferner werden die Möglichkeiten zur Ein- und Ausgabe von Speicherbereichen (BLOAD, BSAVE, BGET, BPUT) aufgezeigt.

Anschließend werden sequentielle (INPUT$, INPUT#, PRINT#) und indexsequentielle (SEEK, RELSEEK) Zugriffsmöglichkeiten sowie Random-Access-Dateien (FIELD, GET#, PUT#, SEEK#, RELSEEK#) vorgestellt. Der Abschnitt Peripherie behandelt die byteweisen Ein- und Ausgabemöglichkeiten wie INP, OUT und deren Nachfragefunktionen INP?, OUT? und den Empfang von der seriellen und der MIDI-Schnittstelle (INPAUX$, INPMID$).

Schließlich geht es um die Maus- und Joystick-Verwaltung (MOUSE, MOUSEX, MOUSEY, MOUSEK, HIDEM, SHOWM, STICK, STRIG) sowie die Ausgabe auf einen Drucker (LPRINT, LPOS, HARDCOPY).

Datenzeilen

DATA konst [,konst1,konst2,...]
READ var [,var1,var2,...]
RESTORE [mar]

```
konst,konst1,konst2: numerische- oder String-Konstante
var,var1,var2: avar oder svar
marke: vom Benutzer definierter Name
```

Mit Hilfe von DATA können konstante Werte platzsparend angegeben werden. Das Einlesen dieser Daten geschieht mit READ. Dabei können numerische Werte in hexadezimaler, oktaler oder binärer Form aufgeführt werden. Wenn eine Zeichenkette eingelesen werden soll, die Kommata enthält, so muß die gesamte Zeichenkette in Anführungsstrichen stehen.

Zu den Befehlen DATA und READ gehört ein sogenannter Datenzeiger. Dieser zeigt immer auf den nächsten mit READ zu lesenden DATA-Wert. Dies ist zu Beginn des Programms der erste Wert hinter dem ersten DATA.

Mit Hilfe des RESTORE-Befehls kann dieser Datenzeiger auf bestimmte DATA-Zeilen gerichtet werden. Zu diesem Zweck wird vor dieser DATA-Zeile eine Marke gesetzt und der Datenzeiger mit RESTORE mar gesetzt. Folgt dem RESTORE-Befehl keine Marke, wird der Datenzeiger auf den ersten im Programm stehenden DATA-Wert gerichtet.

Die Zeichenfolge mar kann aus Ziffern, Buchstaben, Unterstrichen und Punkten bestehen und im Gegensatz zu Variablennamen auch mit einer Ziffer beginnen. Die Marke selbst muß mit einem Doppelpunkt enden.

Beispiel:

```
FOR i=1 TO 3
  READ a
  PRINT a'
NEXT i
'
RESTORE roemische_ziffern
READ a$,b$,c$,d$
PRINT
PRINT a$'b$'c$'d$
'
DATA 1,2,3,4
DATA a,b,c,d
'
roemische_ziffern:
DATA I,II,III,IV
```

--> In einer Schleife werden die numerischen Werte 1,2 und 3 eingelesen und auf dem Bildschirm angezeigt. Danach wird mittels RESTORE der Datenzeiger auf die DATA-Zeile mit den römischen Zahlzeichen gerichtet. Dies geschieht mit roemische_ziffern. Anschließend werden diese Daten den String-Variablen a$,b$,c$ und d$ zugewiesen und ausgegeben.

```
DATA 10,&A,$A,&HA,&O12,&X1010,%1010
FOR i%=1 to 7
  READ a%
  PRINT a%
NEXT i%
```

--> Liest siebenmal die Zahl 10 ein. Bei READ/INPUT/VAL usw. können HEX-Zahlen auch durch $ gekennzeichnet werden und BIN-Zahlen auch durch %, nicht nur durch &H oder & bzw, &X.

Dateiverwaltung

Im Folgenden werden die Befehle erklärt, die der Dateien-Organisation dienen. Dazu muß der Aufbau von Dateispezifikationen nach den Regeln des hierarchischen File-Systems bekannt sein. Eine Dateispezifikation besteht aus drei Teilen: der Laufwerksangabe, dem Dateinamen und der Dateikennung. Die Laufwerksangabe enthält die Laufwerkskennung A bis O, gefolgt von einem Doppelpunkt. Der Dateiname ist 1 bis 8 Zeichen lang. Wahlweise kann eine Dateikennung angegeben werden. Sie besteht aus einem Punkt, gefolgt von 1 bis 3 Zeichen.

Zur weiteren Untergliederung von Dateien gibt es Verzeichnisse (auch Ordner genannt). Diese Verzeichnisse können in verschiedenen Ebenen angelegt werden; die niedrigste Ebene wird auch Wurzelverzeichnis genannt. Ausgehend von diesem Wurzelverzeichnis kann die Verzweigung in weitere Unterverzeichnisse erfolgen. Ein Verzeichniseintrag kann demnach aus den folgenden drei Teilen bestehen:

- Laufwerksangaben
- Namen für Unterverzeichnisse
- Dateinamen und -kennung

Diese Teile werden durch umgekehrte Schrägstriche "\" (Backslash) getrennt. Namen für Verzeichnisse haben das gleiche Format wie Dateinamen. Somit ergibt sich der Zugriffspfad für eine Datei durch Aneinanderfügen der bisher vorgestellten Elemente. Zunächst die Laufwerksangabe, gefolgt von Namen für Unterverzeichnisse, und dann der Dateiname, z.B.:

```
A:\TEXTE.DOC\HANDBUCH\KAPITEL_1.DOC
```

In diesem Beispiel ist der Zugriffspfad also:

A:	Laufwerksangabe
\TEXTE.DOC	Verzeichnis TEXTE.DOC
\HANDBUCH	Unterverzeichnis HANDBUCH
\KAPITEL_1	Dateiname KAPITEL_1
.DOC	Dateikennung .DOC

Zusätzlich stehen zwei Sonderzeichen zur Verfügung, die eine leichtere Dateiauswahl ermöglichen. Sie können innerhalb von Dateinamen und deren Kennungen stehen. Es sind dies das Fragezeichen und der Stern (das Multiplikationszeichen).

Das Fragezeichen ermöglicht die Belegung dieser Position durch jedes zulässige Zeichen, dessen ASCII-Code größer als 32 ist. Es ist auch möglich, das Fragezeichen an mehreren Stellen der Dateispezifikation zu verwenden oder es mehrmals aufeinanderfolgen zu lassen. Der Stern bedeutet, daß jede beliebige Zeichenkette alle restlichen Positionen belegen kann.

Verzeichnisse

DFREE(n)
CHDRIVE n oder n$
DIR$(n)
CHDIR name$

```
n: iexp
name$: sexp
```

DFREE (disk free) liefert den freien Speicherplatz einer Diskettenstation in Bytes. Diese Funktion kann, besonders bei Harddisk-Partionen, sehr lange dauern.

CHDRIVE (change drive) legt das Standard-Laufwerk fest. Wird in einem Ein- oder Ausgabebefehl kein besonderes Laufwerk angegeben, erfolgt die Ein- oder Ausgabe vom bzw. auf das Standard-Laufwerk. Statt der Laufwerksnummer als Zahl kann bei CHDRIVE auch ein String angegeben werden, dessen erstes Zeichen der Laufwerkskennbuchstabe ist.

DIR$(n) ermittelt den aktuellen Zugriffspfad für ein Laufwerk, wie von CHDIR gesetzt.

Für DFREE(n) und DIR$(n) und CHDRIVE n ist der Wert n die Laufwerkskennung. Dabei kann n Werte zwischen 0 und 16 annehmen, wobei 0 dem aktivem Laufwerk und 1 bis 16 den Laufwerken A bis P entsprechen.

CHDIR legt das aktuelle Verzeichnis fest. Da mit CHDIR kein Wechsel des Laufwerks möglich ist, gilt dieser Befehl immer für das aktuelle Laufwerk oder für das angegebene Laufwerk. So wechselt CHDIR "B:\TEST" den Default-Ordner für Laufwerk B:, also den Ordner, der bei Zugriffen auf dieses Laufwerk ohne Pfadangabe, das heißt ohne \, angesprochen wird. Dabei ist der Stringausdruck name$ gleich dem gewünschten Zugriffspfad. Enthält name$ nur einen Rückwärts-Schrägstrich "\", so wird zum Wurzelverzeichnis des aktuellen Laufwerks gewechselt.

Es gibt noch zwei spezielle Ordnernamen "." und "..". Dabei stellt der Ordnername "." eine alternative Bezeichnungsweise für den aktiven Ordner und der Ordnername ".." entsprechend für den übergeordneten Ordner dar.

Wenn es z.B. auf einer Diskette die Ordner \TEST\O1 und \TEST\O2 gibt und der Ordner \TEST\O1 aktiv ist (durch CHDIR), so kann mit CHDIR "..\O2" in den Ordner \TEST\O2 gewechselt werden.

Beispiele:

```
CHDRIVE 1
PRINT DFREE(0)
PRINT DIR$(2)
CHDRIVE "C:\"
```

--> CHDIRVE 1 legt Laufwerk A als Standard-Laufwerk fest. PRINT DFREE(0) gibt freien Speicherplatz (in Bytes) des Standard-Laufwerks (n = 0. Jetzt wegen CHDRIVE 1 das Laufwerk A:) auf dem Bildschirm aus. PRINT DIR$(2) ermittelt den Zugriffspfad auf Laufwerk B:. Anschließend wird Laufwerk C: als Standardlaufwerk festgelegt.

```
CHDIR "\"
CHDIR "TEXTE.DOC\HANDBUCH"
CHDIR "ANHANG"
```

--> CHDIR "\" wechselt zum Wurzelverzeichnis des aktuellen Laufwerks. In der zweiten Zeile wird zum Unterordner HANDBUCH des Verzeichnisses TEXTE.DOC gewechselt. In der dritten Zeile wird weiter in das Unterverzeichnis ANHANG (\TEXTE.DOC\HANDBUCH\ANHANG) verzweigt.

DIR p$ [**TO** name$]
FILES p$ [**TO** name$]

```
p$,name$: sexp
```

Die Befehle DIR und FILES dienen der Ausgabe von Verzeichnissen. Bei der Verwendung von DIR wird das Inhaltsverzeichnis eines bestimmten Laufwerks standardmäßig auf dem Bildschirm ausgegeben. Dabei wird der gewünschte Zugriffspfad im String-Ausdruck p$ angegeben. Gleiches gilt für den FILES-Befehl, jedoch werden dabei zusätzlich Datum, Uhrzeit und Länge der betreffenden Dateien aufgeführt. Außerdem werden dann auch Ordnernamen mit vorangestelltem * ausgegeben, wenn sie die Suchmaske (z.B. *.*) erfüllen, unter Umständen auch die Ordner "." und "..". Wenn die Dateimaske p$ auf ":" oder "\" endet, so hängt das GFA-BASIC selbst "*.*" an (vgl.: FSFIRST und FSNEXT).

Für DIR und FILES ist optional die Angabe des Zusatzes TO name$ möglich. Die Verwendung dieses Zusatzes ermöglicht die Umleitung des Inhaltsverzeichnisses auf eine Datei bzw. ein Peripheriegerät. Der String-Ausdruck name$ enthält dabei den Namen einer Datei bzw. eine Peripheriekennung.

Beispiele:

```
DIR "A:\TEXTVER\*.DOC"
DIR "A:\TEXTVER\HANDBUCH\*.DOC" TO "B:\HANDBUCH\INHALT.ASC"
DIR "A:\*.*" TO "PRT:"
FILES "A:\*.DOC" TO "LST:"
```

--> In der ersten Zeile werden alle Dateinamen mit der Kennung ".DOC" (aus dem Ordner "\TEXTVER" im Laufwerk A) auf dem Bildschirm ausgegeben. Die zweite Zeile bewirkt die Umleitung des Inhaltsverzeichnisses auf eine Datei "INHALT.ASC" im Laufwerk B. Die Ausgabe auf den Drucker geschieht in der dritten Zeile. In der vierten Zeile werden alle Dateinamen mit der Kennung ".DOC" auf den Drucker umgeleitet.

FGETDTA()
FSETDTA(adr)

adr : iexp

Die Funktion FGETDTA() liefert die DTA (disk transfer adress).

Mit FSETDTA wird die DTA gesetzt. Auch eine Fileselect-Box verändert die DTA. Voreingestellt für die DTA ist BASEPAGE+128. Diese Adresse wird auch von DIR, FILES und EXIST benutzt.

Die DTA hat folgenden Aufbau:

Bytes	Offset	Bedeutung
21	0	Für GEMDOS reserviert
1	21	Dateiattribute (siehe unten)
2	22	Uhrzeit
2	24	Datum
4	26	Dateilänge
14	30	Dateiname mit Abschluß durch Nullbyte, ohne Leerstellen.

Die Bedeutung der Attributbits ist:

Bit	Bedeutung
0	Schreibgeschützte Datei.
1	Versteckte Datei.
2	Systemdatei.
3	Diskettenname.
4	Ordner.
5	Archiv-Bit.

FSFIRST(p$,attr)
FSNEXT()

```
p$   : sexp
attr : iexp
```

Die Funktion FSFIRST ermöglicht die Suche nach der ersten Datei, die das in p$ angegebene Suchkriterium (z.B. C:*.GFA) erfüllt. Gefundene Dateinamen und weitere Informationen werden in die DTA geschrieben. Der Parameter attr enthält die Attribute, die die zu findenden Dateien haben dürfen.

Die Funktion FSNEXT() sucht das nächste File, das die Bedingungen von FSFIRST erfüllt.

Beispiel:

```
~FSETDTA(BASEPAGE+128)                    ! setzt DTA
e%=FSFIRST("\*.GFA",-1)                   ! Suchkriterium festlegen
DO UNTIL e%
  IF BYTE{BASEPAGE+149} AND 16            ! wenn es ein Ordner ist
    PRINT "*";CHAR{BASEPAGE+158},         ! Stern davor anzeigen
  ELSE                                    ! sonst ein Leerzeichen
    PRINT 'CHAR{BASEPAGE+158},            ! vor den Dateinamen
  ENDIF
  e%=FSNEXT()                             ! Suche fortsetzen
LOOP
```

--> Gibt alle Dateien mit der Extension .GFA aus dem aktuellen Verzeichnis auf dem Bildschirm aus, sowie alle Ordnernamen, die ebenfalls diese Extension besitzen, durch * markiert.

MKDIR name$
RMDIR name$

```
name$: sexp
```

Der Befehl MKDIR (make directory) legt ein Verzeichnis (Ordner) an. Der String-Ausdruck name$ enthält den zugehörigen Zugriffspfad. Mit RMDIR (remove directory) wird ein Verzeichnis (Ordner) gelöscht, aber nur dann, wenn dieses Verzeichnis keine Unterverzeichnisse oder Dateien enthält.

Beispiel:

```
MKDIR "A:\TEXTVER"
RMDIR "A:\TEXTVER"
```

--> In der ersten Zeile wird ein Verzeichnis "TEXTVER" auf Laufwerk A angelegt, welches in der zweiten Zeile wieder gelöscht wird.

Dateien

EXIST(name$)

```
name$: sexp
```

Mit Hilfe von EXIST kann man feststellen, ob eine Datei vorhanden ist. Dazu wird im String-Ausdruck name$ der Zugriffspfad dieser Datei angegeben. Die Funktion liefert TRUE (-1) zurück, wenn die Datei existiert, sonst FALSE (0) (vgl.: FSFIRST und FSNEXT).

Beispiel:

```
OPEN "U",#1,"TEST.TXT"
PRINT #1,"BEISPIEL"
CLOSE #1
PRINT EXIST("TEST.TXT")
PRINT EXIST("TEST.DOC")
```

--> In den ersten drei Zeilen wird eine Datei mit dem Namen "TEST.TXT" eröffnet. Die vierte Zeile überprüft das Vorhandensein der Datei "TEXT.TXT", es wird TRUE zurückgegeben. In der letzten Zeile ergibt die Überprüfung FALSE, da die Datei nicht existiert. Vgl. auch: FSFIRST, FSNEXT.

OPEN modus$,#n,name$ [anz]

```
modus$,name$: sexp
anz,n: iexp
```

OPEN öffnet einen Datenkanal zu einer Datei oder zu einem Peripheriegerät. Dabei bestimmt der String-Ausdruck "modus$" eine der folgenden Zugriffsmöglichkeiten:

O (output)	Öffnet eine Datei zum Schreiben. Dazu wird sie ggf. neu eingerichtet bzw. bestehende Daten gelöscht.
I (input)	Öffnet eine Datei zum Lesen.
A (append)	Ermöglicht das Anhängen von Daten an eine bestehende Datei. Dabei ist der Datenzeiger auf das Ende der Datei gerichtet.
U (update)	Öffnet eine bereits bestehende Datei zum Schreiben und Lesen.
R (random access)	Öffnet eine Datei zum wahlfreien Schreiben und Lesen. Eine Beschreibung zu diesem Dateityp wird unter dem FIELD-Befehl gegeben.

Der numerische Ausdruck n enthält die Kanalnummer und kann Werte von 0 bis 99 annehmen. Diese Kanalnummer muß beim Arbeiten mit der Datei angegeben werden. Dabei kann das #-Zeichen vor der Kanalangabe entfallen. Der String-Ausdruck name$ enthält den Zugriffspfad der Datei. Als Dateiname kann auch eine Peripheriekennung (siehe Liste bei INP) angegeben werden. Der numerische Ausdruck anz wird nur bei Dateien mit wahlfreiem Zugriff (random access) ausgewertet; er enthält die Länge eines Datensatzes.

Kürzel		Bedeutung	Intern benutzt
LST:	(list)	Paralell	BIOS 0
AUX:	(auxial)	seriell (RS 232)	BIOS 1
CON:	(console)	Tastatur/Bildschirm	BIOS 2 oder VDI
MID:	(musical instruments digital interface)	MIDI-Schnittstelle	BIOS 3
IKB:	(intelligent keyboard)	Tastaturprozessor	BIOS 4
VID:	(video)	Monitor	BIOS 5 oder VDI
PRN:	(printer)	Drucker	GEMDOS -3

LOF(#n)
LOC(#n)
EOF(#n)
CLOSE [#n]
TOUCH [#] n

```
anz, n: iexp
```

Die Funktionen LOF (length of file), LOC (location) und EOF (end of file) können nur auf zuvor mit OPEN geöffnete Dateien angewandt werden. Alle drei Funktionen haben den numerischen Ausdruck "n" gemeinsam. Er gibt die Kanalnummer der Datei an, auf die sich die jeweilige Funktion bezieht. LOF liefert die Länge einer Datei in Bytes. LOC gibt die aktuelle Position des Datenzeigers auch in Bytes (siehe auch SEEK) an. EOF gibt an, ob der Datenzeiger auf das Ende einer Datei zeigt (bzw. die Datei vollständig gelesen wurde). Wenn der Datenzeiger auf das Dateiende zeigt, wird TRUE (-1) zurückgegeben, sonst FALSE (0).

CLOSE schließt einen zuvor mit OPEN geöffneten Datenkanal zu einer Datei oder zu einem Peripheriegerät. Dabei enthält der numerische Ausdruck "n" die Nummer des zu schließenden Kanals. Ohne besondere Kanalangabe werden alle geöffneten Dateien geschlossen.

TOUCH aktualisiert die Zeitangabe einer Datei, das heißt, Datums- und Zeiteintrag einer geöffneten Datei werden auf die Systemdaten gestellt.

Beispiele:

```
OPEN "O",#1,"test.txt"
FOR i%=1 TO 20
  PRINT #1,STR$(i%)
NEXT i%
CLOSE #1
FILES "test.txt"
DELAY 20                ! 20 Sekunden Pause
OPEN "u",#1,"test.txt"
TOUCH #1
CLOSE #1
FILES "test.txt
```

--> Im Beispiel wird die Datei "test.txt" unter der Kanalnummer 1 zum Schreiben geöffnet und geschlossen. Anschließend werden auf dem Bildschirm die Daten dieses Files ausgegeben, nach 20 Sekunden aktualisiert und noch einmal ausgegeben.

```
OPEN "I",#1,"test.txt"
PRINT "   Dateilänge: ";LOF(#1)
PRINT
PRINT "   Daten","Position des Datenzeigers"
DO UNTIL EOF(#1)
  INPUT #1,a$
  PRINT "   ";a$,LOC(#1)
LOOP
CLOSE #1
```

--> Das Beispiel öffnet dieselbe Datei zum Lesen. Zunächst wird die Länge der Datei mittels LOF ausgegeben. Anschließend werden Inhalt der Datei und die zugehörige Datenzeigerposition auf dem Bildschirm ausgegeben. Die Abbruchbedingung für die Schleife wird mit EOF festgelegt. Das Einlesen der Daten geschieht (unabhängig von der zuvor ermittelten Dateilänge) durch Überprüfung des Datenzeigers. Dabei wird die Schleife so oft durchlaufen, bis die Funktion EOF den Wert TRUE (-1) übergibt.

NAME alt$ **AS** neu$
RENAME alt$ **AS** neu$
KILL name$

```
alt$,neu$,name$: sexp
```

NAME benennt eine Datei mit dem Zugriffspfad alt$ in die Datei mit dem Zugriffspfad neu$ um. Dabei wird der Inhalt der Datei nicht verändert. Die Dateien alt$ und neu$ müssen sich auf demselben Laufwerk befinden. RENAME entspricht dem Befehl NAME.

KILL löscht eine Datei, deren Zugriffspfad im String-Ausdruck "name$" angegeben wird.

Beispiel:

```
OPEN "O",#1,"test.txt"
PRINT #1,"Beispiel"
CLOSE #1
'
NAME "test.txt" AS "beispiel.txt"
DIR
KILL "beispiel.txt"
DIR
```

--> Im ersten Beispiel wird die Datei "test.txt" zum Schreiben geöffnet. Anschließend wird diese Datei in "beispiel.txt" umbenannt und schließlich gelöscht. Dies wird durch das Ausgeben des Inhaltsverzeichnisses überprüft. Nach NAME bzw. KILL werden die veränderten Verzeichnisse auf dem Bildschirm angezeigt.

BLOAD n$ [,adr]
BSAVE n$,adr,anz
BGET #n,adr,anz
BPUT #n,adr,anz

```
name$: sexp
n,adr,anz: aexp
```

Mit BSAVE kann man einen Speicherbereich auf Diskette (RAM-Disk, Festplatte usw.) speichern, der z.B. mit BLOAD wieder geladen werden kann. Der numerische Ausdruck adr gibt die Anfangsadresse des Speicherbereiches an. Ohne Angabe von adr wird die bei BSAVE zuletzt genannte Adresse benutzt. Bei BSAVE muß zusätzlich anz (die Länge der Datei n$) angegeben werden. Der Parameter n$ ist der Name der zu ladenden oder zu speichernden Datei. Für n$ gelten die unter DIR beschriebenen Regeln des hierarchischen Dateisystems.

BSAVE, BLOAD können nur gesamte Dateien unter ihrem Namen ansprechen. Im Gegensatz dazu greifen BPUT und BGET auf Dateien über deren Kanalnummer n zu. So ist es mit BGET und BPUT auch möglich, Teile einer Datei (auch mehrfach) zu laden oder zu speichern.

Beispiele:

```
DEFFILL 1,2,4
PBOX 100,100,200,200
BSAVE "rechteck.pic",XBIOS(2),32000
CLS
PRINT AT(4,20);"Bild gespeichert, Taste drücken!"
~INP(2)
CLS
BLOAD "rechteck.pic"
```

--> Zeichnet ein Rechteck und speichert den Bildschirm unter "rechteck.pic" ab. Dann erfolgt eine entsprechende Meldung. Nach einem Tastendruck wird die abgespeicherte Datei wieder in den Bildschirm geladen.

```
DEFFILL 1,2,4
PBOX 0,0,639,199
DEFFILL 1,2,2
PBOX 0,200,639,399
DEFTEXT 1,0,0,32
TEXT 100,115,"Die obere Hälfte"
TEXT 100,315,"Die untere Hälfte"
'
OPEN "O",#1,"screen.pic"
BPUT #1,XBIOS(2),32000
CLOSE #1
PAUSE 25
'
OPEN "I",#1,"screen.pic"
BGET #1,XBIOS(2)+16000,16000
BGET #1,XBIOS(2),16000
CLOSE #1
```

--> Füllt die obere Bildschirmhälfte mit einem Muster und die untere Bildschirmhälfte mit einem anderen. Danach wird der gesamte Bildschirm in der Datei screen.pic gespeichert und nach einer Pause wieder eingelesen. Dabei wird die erste Hälfte der Datei in die untere Bildschirmhälfte und die zweite Hälfte in die obere Hälfte des Bildschirms geladen.

Sequentieller Zugriff

INP(#n)
OUT #n,a [,b,c,...]

```
n,a,b,c,...: iexp
```

INP(#n) liest ein Byte aus einer zuvor geöffneten Datei. Analog übergibt OUT ein Byte an eine Datei. Dabei enthält der numerische Ausdruck "n" die Kanalnummer (0 bis 99), unter der die entsprechende Datei angesprochen wird. INP und OUT ohne "#" können auch für die Kommunikation mit Bildschirm, IKBD, MIDI-Schnittstelle usw. verwendet werden (siehe dort).

Bei OUT wird BYTE(a) verwendet.

Beispiel:

```
OPEN "O",#1,"test.txt"
OUT #1,128
CLOSE #1
'
OPEN "I",#1,"test.txt"
a=INP(#1)
CLOSE #1
PRINT a
```

--> Im ersten Beispiel wird eine Datei zum Schreiben geöffnet und ein Byte geschrieben, welches im zweiten Teil des Beispiels in die numerische Variable "a" eingelesen und auf den Bildschirm ausgegeben wird. Dabei hat "a" den Wert 128.

INPUT$(anz [,#n])

n, anz: iexp

INPUT$ liest eine Zeichenkette, die aus anz Zeichen besteht, von der Tastatur. Optional bewirkt die Angabe einer Kanalnummer "n" (0 bis 99) das Einlesen von Zeichen aus einer Datei. In beiden Fällen gibt der numerische Ausdruck "anz" an, wie viele Zeichen gelesen werden.

Beispiel:

```
OPEN "O",#1,"version.dat"
PRINT #1,"GFA-BASIC, Version 3.0"
CLOSE #1
'
OPEN "I",#1,"version.dat"
v$=INPUT$(9,#1)
CLOSE #1
PRINT v$
PRINT "Geben Sie bitte die Versionsnummer an: ";
PRINT INPUT$(3)
```

--> Im ersten Beispiel wird die Datei "version.dat" geöffnet und eine Meldung in die Datei geschrieben. Der zweite Teil des Beispiels liest die ersten 9 Zeichen dieser Datei in die String-Variable v$ ein und gibt sie anschließend auf den Bildschirm aus. Danach erscheint eine Meldung, und es werden 3 Zeichen von der Tastatur eingelesen und auf den Bildschirm ausgegeben.

INPUT #n,var1 [,var2,var3,...]
LINE INPUT #n,a1$ [,a2$,a2$,...]

```
n: iexp
a1$, a2$, a3$: sexp
var1, var2, var3: avar oder svar
```

INPUT #n ermöglicht das Einlesen von Daten aus einer Datei. Dabei können einzelne Werte oder Variablenlisten eingelesen werden, wobei die Variablen durch Kommata getrennt werden. Diese Befehle entsprechen INPUT bzw. INPUT LINE, nur wird hier (meist) nicht von der Tastatur gelesen.

Beispiel:

```
OPEN "I",#1,"TEXT.DOC"
INPUT #1,a$,b$
LINE INPUT #1,c$
CLOSE #1
PRINT a$
PRINT b$
PRINT c$
```

--> Liest drei Strings aus einer Datei (die vorher geöffnet werden muß) und gibt sie auf den Bildschirm aus.

PRINT #n,ausdruck
PRINT #n,**USING** form$,ausdruck
WRITE #n,ausdruck

```
n: iexp
form$: sexp
ausdruck: aexp oder sexp oder deren Kombinationen
```

PRINT #n gibt Daten auf einen Datenkanal aus. PRINT #n, USING erlaubt die formatierte Ausgabe auf einen Datenkanal. In beiden Fällen steht dabei n für die Kanalnummer (0 bis 99) der entsprechenden Datei. Ansonsten wie PRINT, PRINT USING und WRITE. Allerdings ist PRINT #n,AT(,) nicht möglich.

Der Befehl WRITE dient in erster Linie der platzsparenden Datenspeicherung in sequentielle Dateien und ist auf das Einlesen von Dateien mit dem INPUT-Befehl zugeschnitten. Die Ausdrücke werden durch Kommata getrennt und Zeichenketten müssen in Anführungszeichen eingeschlossen werden.

Beispiele:

```
OPEN "O",#1,"TEXT.DOC"
b$="wort"
PRINT #1,"Test",a$
PRINT #1,"GFA-","BASIC"
CLOSE #1
```

--> Liest drei Strings aus einer Datei (die vorher eingerichtet werden muß) und gibt sie auf dem Bildschirm aus.

```
OPEN "O",#1,"TEST.DAT"
WRITE #1,"Version",3,".0"
CLOSE #1
'
OPEN "I",#1,"TEST.DAT"
INPUT #1,v1$,v2$,v3$
CLOSE #1
'
PRINT v1$+v2$+v3$
```

--> Schreibt Daten durch Kommata getrennt in die Datei TEST.DAT, liest anschließend die Daten mit INPUT wieder ein und gibt sie auf den Bildschirm aus.

STORE #i,x$() [,n]
RECALL #i,x$(),n,x

```
i,n: iexp
x$(): String-Array
x: Variable, mindestens 32 Bit
```

Der Befehl STORE dient zum Abspeichern eines String-Feldes als Text-File (mit Trennung durch CR/LF). Es wird das komplette String-Array über den geöffneten Kanal i ausgegeben. Der wahlfreie Parameter n kann angeben, wieviele Elemente des String-Arrays ausgegebene werden sollen.

Der Befehl RECALL dient zum schnellen Einlesen eines String-Feldes aus einem Text-File. Es werden n Zeilen des Text-Files in das String-Array eingelesen. Wenn n zu groß für die Dimensionierung des String-Arrays ist, so wird die einzulesende Anzahl automatisch begrenzt (n = -1 liest das gesamte Array). Wird während des Lesens das File-Ende (EOF) erreicht, so wird das Einlesen ohne Fehlermeldung abgebrochen. In jedem Fall enthält die Variable x am Ende die tatsächlich gelesene Anzahl Strings.

Beispiele:

```
DIM A$(1000)
FOR i%=0 TO 499
  a$(i%)=STR$(RND)     !irgendetwas
NEXT i%
OPEN "O",#1,"testfile.txt"
STORE #1,a$(),500
CLOSE #1
'
DIM b$(2000)
OPEN "I",#1,"testfile.txt"
RECALL #1,b$(),-1,n
CLOSE #1
PRINT n
```

--> Es wird die Anzahl der gelesenen Textzeilen ausgegeben, hier also 500.

```
PRINT "Zeilenzähler"
DIM a$(1000)
DO
  FILESELECT "\*.*","",f$
  EXIT IF f$=""
  lc%=0
  OPEN "I",#1,f$
  DO
    RECALL #1,a$(),-1,x%
    ADD lc%,x%
  LOOP WHILE x%
  CLOSE #1
  PRINT f$;" enthält ";lc%;" Zeilen"
LOOP
```

--> Dieses Programm zählt die Zeilen in Text-Files.

Anmerkung: STORE funktioniert auch für zeichenorientierte Dateien (AUX:), RECALL dagegen nicht, da intern ein SEEK verwendet wird.

SEEK #n,pos
RELSEEK #n,anz

```
n,anz,pos: iexp
```

Die Befehle SEEK und RELSEEK erlauben die Positionierung des Datenzeigers. Dadurch läßt sich ein indexsequentieller Dateizugriff realisieren. Dabei enthält der numerische Ausdruck "n" die Kanalnummer einer zuvor mit OPEN geöffneten Datei. Beide Befehle können nur bei Dateien, nicht aber bei Peripheriegeräten verwendet werden. Der Datenzeiger gibt an, welches Byte einer Datei zuletzt gelesen oder geschrieben wurde. Außer beim Modus "A" (Datenanfügen an eine bestehende Datei) hat der Datenzeiger beim Öffnen einer Datei den Wert 0. Die Schreib- und Lesebefehle werden mit dem ersten Byte begonnen, auf das der Datenzeiger gerichtet ist.

Der absoluten Positionierung des Datenzeigers dient der SEEK-Befehl. Dazu wird der Datenzeiger auf das in "pos" angegebene Byte gerichtet. Die relative Zeigerpositionierung erfolgt mit dem RELSEEK-Befehl. Dabei wird der Datenzeiger von seiner aktuellen Position um den in "anz" angegebenen Wert verschoben. RELSEEK ist meistens schneller!

Die numerischen Ausdrücke "anz" und "pos" dürfen nur Werte von 0 bis zur entsprechenden Dateilänge annehmen. Bei positiven Werten wird der Datenzeiger in Richtung Dateiende und bei negativen Werten in Richtung Dateianfang bewegt. Bei der Angabe von 0 wird der Datenzeiger auf den Dateianfang gerichtet.

Beispiel:

```
OPEN "O",#1,"X.X"
PRINT #1,STRING$(20,42)
SEEK #1,10
PRINT #1,"#";
RELSEEK #1,-5
OUT #1,33,48
CLOSE #1
OPEN "I",#1,"X.X"
LINE INPUT #1,a$
PRINT a$
CLOSE§ #1
```

--> Ausgabe: ******!0**#*********

Wahlfreier Zugriff

Im Folgenden wird der Umgang mit einer sogenannten Random-Access-Datei (Datei mit wahlfreiem Zugriff) erläutert. Dabei sind zwei Begriffe besonders wichtig: Datensatz und Datenfeld. Ein Datensatz stellt im Folgenden eine logische Zusammenfassung von Daten dar, z.B. ein Datensatz "adressen". Dieser Datensatz kann in Datenfelder unterteilt werden, z.B. "name", "strasse". Analog werden die Begriffe Datensatz- und Feldlänge verwendet; sie geben die Länge des Datensatzes bzw. der Datenfelder in Bytes an.

Der wesentliche Unterschied zwischen einer Random-Access-Datei und einer sequentiellen Datei besteht im Datenzugriff. Bei einer sequentiellen Datei muß die gesamte Datei geladen werden, um auf einen Datensatz zugreifen zu können. Jedoch kann ein Datensatz aus einer Random-Access-Datei gelesen werden, ohne die gesamte Datei einlesen zu müssen. Dies ist besonders bei sehr großen Dateien nützlich. Jedoch muß dieser Vorteil mit einem größeren Speicherplatzbedarf auf der Diskette erkauft werden, denn eine Datei mit wahlfreiem Zugriff arbeitet mit festen Datensatz- und Feldlängen. Unabhängig von der Länge der Daten werden diese Satz- und Feldlängen abgespeichert.

FIELD #n,anz **AS** satz$ [, anz **AS** satz$, anz **AS** satz$,...]
FIELD #n,anz **AT**(x) [,anz **AT**(x) [,anz **AT**(x) [,,...]

```
n, anz,x: iexp
satz$: svar, jedoch keine Feldvariable
```

Der Befehl FIELD AS unterteilt Datensätze in Felder. Der numerische Ausdruck n ist die Nummer des Datenkanals (0 bis 99) einer mit OPEN geöffneten Datei. Der ganzzahlige Ausdruck anz bestimmt die Feldlänge. Die String-Variable satz$ nimmt ein Feld eines Datensatzes auf. Soll der Datensatz in mehrere Felder aufgeteilt werden, müssen die Befehlsteile (anz AS satz$) durch Kommata getrennt werden. Die Summe der einzelnen Feldlängen muß der Datensatzlänge entsprechen, sonst wird eine entsprechende Fehlermeldung ausgegeben. Damit die einzelnen Datensätze keine andere Länge haben als im FIELD-Befehl festgelegt, ist es zweckmäßig, die Befehle LSET und RSET oder MID$ zu benutzen.

Mit Hilfe von AT können z.B. numerische Variablen in eine R-Datei (Random Access) geschrieben werden, ohne daß sie in Zeichenketten überführt werden müssen, indem in den Klammern ein Zeiger auf die abzuspeichernde numerische Variable und vor AT die Anzahl der Bytes, die ab dieser Adresse gelesen und abgespeichert werden soll, steht. Es können auch beliebige andere Adressen angegeben werden. Zum Beispiel:

```
FIELD #1,4 AT(*a%),2 AT(*b&),8 AT(*c#)
```

Außerdem sind beliebige Kombinationen von AS und AT möglich, z.B.:

```
FIELD #2,4 AS a$,2 AT(*b&),8 AT(*c#),6 AS d$
```

Im Unterschied zur Version 2.0 kann ein FIELD-Befehl auf mehrere Zeilen aufgeteilt werden. Die maximale Satzlänge beträgt 32767, die maximale Feldanzahl beträgt dabei etwa 5000.

GET #n [,satz]
PUT #n [,satz]
RECORD #n,satz

```
n,satz: iexp
```

GET liest einen Datensatz aus einer "R"-Datei. Analog speichert PUT einen Datensatz in eine solche Datei. Dabei ist n die Nummer (0 bis 99) eines mit OPEN geöffneten Datenkanals. Der optionale Parameter satz enthält einen Wert zwischen 1 und der Anzahl der Datensätze innerhalb der entsprechenden Datei. Dieser gibt die Nummer des zu lesenden oder zu speichernden Datensatzes an. Ohne Angabe von satz wird der jeweils nächste Datensatz gelesen oder gespeichert.

Mit RECORD wird nur die nächste Satznummer für PUT oder GET gesetzt. Nach RECORD #1,15 liest GET #1 also Satz Nr. 15.

Achtung: Eine Datei kann immer nur um einen Datensatz verlängert werden, gegebenenfalls bei Anlegen in einer Schleife mehrere Datensätze anlegen.

Beispiele:

```
OPEN "R",#1,"\privat.rdm",62
FIELD #1,24 AS name$,24 AS strasse$,2 AT(*plz&),12 AS ort$
'
FOR i%=1 TO 3
  INPUT "Name   : ";n$
  INPUT "Strasse: ";s$
  INPUT "PLZ    : ";plz&
  INPUT "Ort    : ";o$
  LSET name$=n$
  LSET strasse$=s$
  LSET ort$=o$
  PUT #1,i%
```

```
  CLS
NEXT i%
'
CLOSE #1
```

--> Zunächst wird eine Random-Access-Datei (Modus "R") mit einer Satzlänge von 62 Bytes eröffnet. Mit Hilfe des FIELD-Befehls wird ein Datensatz in 2 Datenfelder mit je 24, ein Feld mit 2 und ein Datenfeld mit 12 Bytes unterteilt. Die Summe der Datenfelder (24+24+2+12) ergibt wieder 62 (also die beim OPEN-Befehl angegebene Datensatzlänge). Anschließend werden die Adreßdaten eingegeben und linksbündig in die entsprechenden Variablen eingesetzt. Danach wird der gesamte Datensatz in die Datei geschrieben. In älteren GFA-BASIC-Versionen stände statt 2 AT(*plz%) etwa 2 AS plz$ und dann vor dem PUT: LSET plz$=MKI$(plz%) und entsprechend nach GET: plz%=CVI(plz$)

```
OPEN "R",#1,"\privat.rdm",62
FIELD #1,24 AS name$,24 AS strasse$,2 AT(*plz&),12 AS ort$
'
FOR i%=1 TO 3
  GET #1,i%
  PRINT "Datensatznummer: ";i%
  PRINT "Name   : ";name$
  PRINT "Strasse: ";strasse$
  PRINT "PLZ    : ";plz&
  PRINT "Ort    : ";ort$
NEXT i%
'
CLOSE #1
```

--> Hierbei wird die Datei mit wahlfreiem Zugriff "PRIVAT_1.RDM" im Unterordner "ADRESSEN" geöffnet und 3 Datensätze gelesen.

Kommunikation mit der Peripherie

Byteweise Ein- und Ausgabe

INP(n)
INP?(n)
OUT [#]n,a [,b...]
OUT?(n)

```
n,a,b: iexp
```

INP liest ein Byte von der Peripherie. Dabei kann der numerische Ausdruck "n" Werte von 0 bis 5 (siehe untenstehende Tabelle) annehmen oder eine Kanalnummer enthalten. Der Befehl OUT übergibt Bytes an die Peripherie. Im Gegensatz zu den GFA-BASIC-Versionen 2.xx kann OUT nun mehrere Bytes an die Peripherie bzw. an eine Datei übergeben.

INP? bzw. OUT? ermittelt den Ein- bzw. Ausgabestatus eines Peripheriegerätes. Dabei wird eine Zahl ungleich Null zurückgemeldet, wenn das unter n angegebene Gerät bereit ist, sonst FALSE (0).

n	Kürzel		Bedeutung
0	LST:	(list)	Drucker
1	AUX:	(auxial)	seriell (RS 232)
2	CON:	(console)	Tastatur
3	MID:	(musical instruments digital interface)	MIDI-Schnittstelle
4	IKB:	(intelligent keyboard)	Tastaturprozessor
5	VID:	(video)	Monitor

Beispiele:

```
PRINT AT(4,4);"Bitte eine Taste drücken!"
~INP(2)
```

--> Gibt eine Meldung auf dem Bildschirm aus und wartet auf das Drücken einer Taste (Peripherienummer 2, Tastatur).

```
OUT 2,27,69,13
```

--> Entspricht dem CLS-Befehl, löscht den Bildschirm und setzt den Cursor in die linke obere Ecke mit Hilfe der VT-52-Sequenz Clear Screen (esc + "E").

Serielle- und MIDI-Schnittstelle

INPAUX$
INPMID$

Mit Hilfe von INPAUX$ und INPMID$ können sehr schnell Daten von der seriellen und der MIDI-Schnittstelle gelesen werden

Beispiel:

```
DO
  PRINT INPAUX$;
LOOP UNTIL MOUSEK
```

--> Liest sehr schnell alle Daten aus dem Input-Puffer der seriellen Schnittstelle ein. Diese Art der Dateneingabe ist sehr viel schneller als das byteweise Einlesen von der Schnittstelle.

```
inp_aux$ = ""
WHILE INP?(1)
  inp_aux$ = inp_aux$ + CHR$(INP(1))
WEND
```

--> Diese Variante liest die Daten deutlich langsamer ein.

Maus und Joystick

MOUSEX
MOUSEY
MOUSEK
MOUSE mx,my,mk

```
mx,my,mk: avar
```

Die Befehle MOUSEX, MOUSEY und MOUSEK ermöglichen die Abfrage der aktuellen Mausposition und des Maustastenstatus. Mit Hilfe des Befehls MOUSE kann man diese Abfragen zusammenfassen. Dann liefert MOUSE in mx und my die aktuellen Mauskoordinaten und in mk den Maustastenstatus. Dabei kann die Variable mk Werte zwischen 0 und 3 annehmen.

mk	Gedrückte Taste
0	Keine
1	Linke
2	Rechte
3	Linke und rechte

Beispiel:

```
REPEAT
  IF MOUSEK=1
    PLOT MOUSEX,MOUSEY
  ENDIF
UNTIL MOUSEK=2
'
REPEAT
  MOUSE mx%,my%,mk%
  IF mk%=2
    PLOT mx%,my%
  ENDIF
UNTIL mk%=1
```

--> In der REPEAT-Schleife werden Mausposition und Maustastenstatus abgefragt. Nur während die linke Maustaste gedrückt ist, wird ein Punkt an der Stelle mx,my gezeichnet. Die Schleife wird nach Drücken der rechten Maustaste verlassen. In der zweiten Schleife kann dann mit der rechten Maustaste gezeichnet und mit der linken das Programm verlassen werden.

SETMOUSE mx,my[,mk]

mx,my,mk: iexp

Die Anweisung SETMOUSE erlaubt die programmgesteuerte Positionierung des Maus-Cursors. Der optionale Parameter mk kann entweder das Drücken oder Loslassen der Maustasten simulieren. Dies gilt leider nur für das VDI, nicht (oder nur selten) für das AES.

Beispiel:

```
FOR i%=00 TO 300
  SETMOUSE i%,i%
  PAUSE 2
  PLOT MOUSEX,MOUSEY
  SHOWM
NEXT i%
```

--> Bewegt den Maus-Cursor diagonal über den Bildschirm und setzt dabei Punkte.

HIDEM
SHOWM

Die Befehle HIDEM und SHOWM bewirken das Ab- bzw. Anschalten des Maus-Cursors. Der Aufruf eines Alert-Befehls oder einer anderen AES-Routine bewirkt, wie die Mausabfragebefehle, ein Einschalten der Maus.

Bei Bildschirmausgaben (PRINT), Grafikbefehlen (VDI, LINE-A) wird dieser Mauscursor abgeschaltet.

Durch HIDEM kann das automatische Wiedereinschalten des Maus-Cursors durch Bewegung (zwischen Befehlen) blockiert werden.

Beispiel:

```
REPEAT
  IF MOUSEK=1
    SHOWM
  ENDIF
  IF MOUSEK=2
    HIDEM
  ENDIF
UNTIL MOUSEK=3
```

--> Durch das Drücken der linken Maustaste wird der Maus-Cursor angezeigt, nach dem Drücken der rechten Taste vom Bildschirm entfernt. Gleichzeitiges Drücken beider Maustasten beendet das Programm.

STICK m
STICK(p)
STRIG(p)

```
m,p: iexp
```

Der ATARI ST verfügt über zwei Schnittstellen (Ports) zum Anschluß von Maus und Joystick. Am Port 0 können Maus- oder Joystick-Koordinaten abgefragt werden, am Port 1 nur Joystick-Daten.

Der Befehl STICK 0 schaltet in einen Modus, in dem von Port 0 Mauskoordinaten zurückgemeldet werden. Für STICK 1 werden von Port 0 und 1 Joystick-Koordinaten zurückgemeldet. Normalerweise ist die Verwendung des Befehls STICK nicht notwendig, da bei Mauskoordinatenabfrage automatisch ein STICK 0 und bei Joystick-Abfrage ein STICK 1 durchgeführt wird. Bei Joystick-abfragenden Programmen sollte aber vor jedem Aufruf von AES-Funktionen (Alert-, Dialogboxen) ein STICK 0 erfolgen.

Die Funktion STICK(p) dient der Abfrage der Joystick-Position. Für p = 0 wird die Position des Joysticks in Port 0, für p = 1 die des Joysticks in Port 1 gemeldet.

```
5    1    9
   \ | /
4 -- 0 -- 8
   / | \
6    2   10
```

Entsprechend ermittelt STRIG(p) den Staus des Feuerknopfes (TRUE = gedrückt, sonst FALSE).

Beispiele:

```
STICK 1              !Joystick-Koordinaten abfragen,
'                     wenn Joystick an Port 1 angeschlossen
REPEAT
  richtung%=STICK(1)
  feuer!=STRIG(1)
  SELECT richtung%
  CASE 4
    PRINT "Links"
  CASE 8
    PRINT "rechts"
  CASE 2
    PRINT "unten"
  CASE 1
    PRINT "oben"
  ENDSELECT
UNTIL feuer!
WHILE STRIG(1)
WEND                  ! Wartet auf das Loslassen des Feuerknopfes
```

--> Durch das Umschalten auf die Maus werden die Meldungen des Joysticks abgeschaltet, d.h. der Rechner bemerkt nicht, daß der/die Feuerknöpfe nicht gedrückt sind. Zur Abhilfe kann man (wie im Beispiel) am Programmende auf das Loslassen beider Feuerknöpfe warten oder das Programm zu Beginn auf eine Änderung des Feuerknopfstatus warten lassen. Beispiel s.u.

```
PRINT "Bitte Feuerknopf drücken"
WHILE STRIG(0)
WEND
REPEAT
UNTIL STRIG(0)
WHILE STRIG(0)
WEND
```

--> Ergänzung zum obigen Beispiel.

Drucker

LPRINT ausdruck
LPOS(x)
HARDCOPY

```
ausdruck: aexp oder sexp oder deren Kombinationen
x: avar, Scheinargument
```

LPRINT ausdruck ermöglicht die Ausgabe von Daten an den Drucker. Dabei ist LPRINT mit dem PRINT-Befehl identisch. Grundsätzlich können alle PRINT-Befehlsvariationen verwendet werden. Es ist aber nicht möglich, mit PRINT AT(x,y) den Druckkopf zu positionieren, aber LPRINT USING ist möglich.

LPOS(x) liefert die Anzahl der seit dem letzten CR an den Drucker ausgegebenen Zeichen (vgl. POS(x)).

HARDCOPY bewirkt eine Ausgabe des Bildschirminhaltes an einen geeigneten Drucker. Dabei wird die Hardcopy-Routine des Betriebssystems aufgerufen. Diese Routine kann auch durch gleichzeitiges Drücken von Alternate- und Help-Taste aufgerufen werden. Es gibt Treiber-Programme für nicht Epson-kompatible 9-Nadel-Drucker, die nicht auf den HARDCOPY-Befehl (XBIOS(20)) reagieren. Dann hilft SDPOKE &H4EE,0.

Beispiele:

```
LPRINT
LPRINT "Test"
PRINT LPOS(x)
```

--> Führt auf dem Drucker (wenn ein solcher angeschlossen und eingeschaltet ist) einen Zeilenvorschub aus, schreibt das Wort "Test" auf dem Drucker und zeigt die aktuelle Position des Druckkopfes auf dem Bildschirm an.

```
FOR i%=20 TO 180 STEP 10
  CIRCLE 320,200,i%
NEXT i%
HARDCOPY
```

--> Zeichnet konzentrische Kreise auf dem Bildschirm und gibt dieses Bild auf dem Drucker aus.

Tonerzeugung

SOUND kan,laut,note,oktave,verz
SOUND kan,laut,#per,verz
WAVE stim,huell,form,dauer,verz

```
kan,laut,note,oktave,verz,per,stim,huell,form,dauer: iexp
```

Mit Hilfe der Befehle SOUND und WAVE wird der dreistimmige Tongenerator des ATARI ST gesteuert. Dazu wird bei SOUND als erster Parameter der Ausdruck "kan" benutzt; er gibt den gewünschten Kanal (1 bis 3) an. Die Lautstärke (1 bis 15) wird über den zweiten Parameter "laut" gesteuert. Die Parameter "note" und "oktave" dienen der Festlegung einer Note (1 bis 12) im Oktavenbereich (1 bis 8). Dabei gilt:

Note	**C**	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
Ziffer	1	2	3	4	5	6	7	8	9	10	11	12

Eine Variante des SOUND-Befehls erlaubt die Angabe einer Schwingungsperiode mit Hilfe des numerischen Ausdrucks "per", dem jedoch das #-Zeichen vorangestellt werden muß. Dabei kann "per" einen Wert von 0 bis 4095 annehmen und läßt sich, wenn die Frequenz in Hertz bekannt ist, wie folgt errechnen:

```
per=ROUND(125000/frequenz)
```

So kann der Kammerton A (440 Hz) auf zwei Arten erzeugt werden:

```
SOUND 1,15,10,4,250
SOUND 1,15,#284,250
SOUND 1,0,0,0,0
```

In beiden Befehlsvarianten folgt ein letzter Parameter "verz"; er gibt die Zeit in 1/50 Sekunden bis zur nächsten Befehlsausführung an (im Beispiel 5 Sekunden). Die Tonerzeugung wird durch den Befehl SOUND 1,0,0,0,0 (wie im Beispiel) oder irgendeinen anderen SOUND-Befehl beendet, wie

ihn z.B. das Betriebssystem bei Tastendruck erzeugt. Dieser Tastenklick kann ausgeschaltet werden durch:

```
SPOKE &H484, BCLR(PEEK(&H484),0)
```

Er kann eingeschaltet werden durch:

```
SPOKE &H484, BSET(PEEK(&H484),0)
```

Die Erzeugung mehrstimmiger Töne ist mittels WAVE möglich. In "stim" werden die Bit-Werte der drei Stimmen addiert und die Frequenz des Rauschgenerators festgelegt:

```
 1 = Stimme 1.
 2 = Stimme 2.
 4 = Stimme 3.
 8 = Rauschen für die erste Stimme.
16 = Rauschen für die zweite Stimme.
32 = Rauschen für die dritte Stimme.
```

Dazu kann das 256-fache der Periode des Rauschgenerators (0 bis 31) addiert werden. In "huell" wird wie bei "stim" bitweise angegeben, welche Stimme durch eine Hüllkurve moduliert werden soll. Die Hüllkurvenform kann durch "form" wie folgt bestimmt werden:

```
0 bis 3 = Linear fallend.
4 bis 7 = Linear steigend, abbrechend.
      8 = Sägezahn fallend.
      9 = Linear fallend (wie 0 bis 3).
     10 = Dreieck, Beginn fallend.
     11 = Linear fallend, sprunghaft laut.
     12 = Sägezahn steigend.
     13 = Linear steigend, haltend.
     14 = Dreieck, Beginn steigend.
     15 = Linear steigend, abbrechend (wie 4 bis7).
```

Die Periode der Hüllkurve wird durch "dauer" festgelegt. Wie beim SOUND-Befehl legt "verz" die Zeit vor der nächsten Befehlsausführung in 1/15 Sekunden fest.

Für SOUND und WAVE gilt: Parameter, die am Befehlsende stehen und nicht verändert werden sollen, können entfallen.

Beispiel:

```
SOUND 1,15,1,4,20
SOUND 2,15,4,4,20
SOUND 3,15,8,4,20
WAVE 7,7,0,65535,300
```

--> Mit jedem Tonkanal wird ein Ton erzeugt und mit Hilfe von WAVE moduliert.

7 - Programmsteuerung

In diesem Kapitel werden alle Befehle zur Steuerung eines Programms besprochen. Es beginnt mit Entscheidungsanweisungen, die eine bedingte Abarbeitung von Befehlen ermöglichen (IF, THEN, ELSE, ENDIF, ELSE IF). Bei diesen Befehlen wird also lediglich geprüft, ob eine bestimmte logische Bedingung wahr oder falsch ist.

Damit verwandt sind die Mehrfachentscheidungsanweisungen (SELECT, CASE, DEFAULT, ENDSELECT, CONT). Bei diesen Befehlen kann das Entscheidungskriterium nicht nur wahr oder falsch sein, sondern beliebige Werte annehmen, auf die selektiv reagiert werden kann.

Der nächste Abschnitt stellt die Schleifenbefehle des GFA-BASIC vor, welche die wiederholte Ausführung von Programmteilen ermöglichen. GFA-BASIC 3.0 verfügt über eine sehr große Zahl solcher Schleifenarten (FOR TO STEP NEXT, REPEAT UNTIL, DO LOOP bzw. ENDLOOP, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL, EXIT IF).

Für die strukturierte Programmierung ist die Möglichkeit der Unterprogrammbildung von großer Wichtigkeit. Es können mit den Befehlen PROCEDURE, GOSUB (@) und RETURN bzw. ENDSUB neue Befehle in Form von Unterroutinen erstellt werden. Die Bildung von Funktionen ist ebenfalls möglich. Mit den Befehlen DEFFN und FN lassen sich Funktionen bilden, die den Charakter von Formeln haben. Noch flexiblere Möglichkeiten ergeben sich mit den Befehlen FUNCTION, ENDFUNC und RESULT, mit denen sich komplette Unterprogramme bilden lassen, die einen Wert zurückliefern.

Ebenfalls in diesen Abschnitt gehört die Variablenverwaltung bei Unterroutinen. Hier gibt es zunächst die Möglichkeit der Vereinbarung lokaler Variablen (LOCAL), die nur in einem Teil des Programms bekannt sind. Variablen können an Unterroutinen als Wert (call by value) übergeben werden. Mit Hilfe der Anweisung VAR können sie aber auch selbst übergeben werden (call by reference), so daß sie von der Unterroutine auch

verändert werden können, ohne daß ihr globaler Variablenname benutzt werden muß.

Im Abschnitt über ereignisbedingte Verzweigungen geht es um zwei GFA-BASIC spezifische Ereignisse. Das erste ist das gleichzeitige Drücken der Control-, Alternate- und der linken Shift-Taste, was normalerweise zum Programmabbruch führt. Das zweite Ereignis ist das Auftreten eines Fehlers bei der Programmausführung. Hier werden die Möglichkeiten zur Programmierung einer Fehlerbehandlungsroutine besprochen.

Es ist in GFA-BASIC 3.0 möglich, eine Unterroutine nach Ablauf einer bestimmten Zeit aufzurufen. Diesem Zweck dienen die Befehle EVERY und AFTER.

Am Ende dieses Kapitels werden die unbedingte Verzweigung GOTO, die Wartebefehle PAUSE und DELAY sowie verschiedene Möglichkeiten des Programmabbruchs (QUIT, SYSTEM, END, EDIT, NEW, STOP) vorgestellt.

Der letzte Abschnitt hat die Befehle zur Überwachung des Programmlaufs (TRON, TROFF, TRON proc, TRACE$ DUMP, ERR$, ERROR) zum Thema.

Entscheidungsanweisungen

IF bed [**THEN**]
ELSE
ENDIF

```
bed: bexp
```

Diese Kommandos ermöglichen es, einen oder mehrere Befehle nur dann wirksam werden zu lassen, wenn eine logische Bedingung erfüllt ist. Folgendes Beispiel soll dies erläutern:

```
IF a=1 THEN
  PRINT "a ist gleich 1"
  b=2
ENDIF
```

In diesem Fall ist a = 1 die logische Bedingung. Die Befehle in den Zeilen zwischen IF und ELSE werden nur abgearbeitet, wenn diese logische Bedingung wahr ist. Ist sie unwahr, so wird der Programmlauf hinter dem Befehlswort ENDIF fortgesetzt. Das Kommando THEN wird nicht beachtet, es genügt also auch die Formulierung:

```
IF a=1    statt    IF a=1 THEN
```

Etwas komplexer ist folgende Konstruktion:

```
IF a=1
  PRINT "a ist gleich 1"
ELSE
  PRINT " a ist nicht gleich 1,"
  PRINT " sondern gleich ";a
ENDIF
```

In diesem Fall werden die Befehle zwischen IF und ELSE abgearbeitet, wenn die logische Bedingung hinter IF wahr ist. Anschließend fährt das Programm hinter dem Kommando ENDIF fort.

Ist die Bedingung hinter IF aber nicht erfüllt, so werden die Befehle zwischen ELSE und ENDIF wirksam. Anschließend wird die Programmausführung wieder hinter ENDIF fortgesetzt. Logisch wahr ist jeder numerische Ausdruck, der ungleich 0 ist.

Beispiel:

```
x=1
IF x
  PRINT "x ist wahr"
ENDIF
INPUT y
IF x=9 OR ODD(y)
  PRINT "y ist eine ungerade Zahl"
ELSE
  PRINT "y ist eine gerade Zahl"
ENDIF
```

--> Zunächst erscheint der Text 'x ist wahr'. Dann wird eine Zahl abgefragt. Da x nicht 9 sein kann, erscheint der Text 'y ist eine ungerade Zahl', wenn Sie eine ungerade Zahl eingegeben haben, andernfalls wird 'y ist eine gerade Zahl' auf den Monitor geschrieben.

ELSE IF bed

bed: bexp

Der Befehl ELSE IF ermöglicht es, verschachelte IF-Anweisungen übersichtlicher darzustellen. Das folgende Beispiel reagiert auf das Drücken von Tasten. Bei S, L oder E wird ein Platzhalter für eine Speicher-, Lade- oder Eingaberoutine aufgerufen. In allen anderen Fällen erscheint der Text 'Unbekanntes Kommando'. Die verschachtelte Version dieses Beispiels lautet dann:

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="L"
    PRINT "Laden"
  ELSE
    IF t$="S"
      PRINT "Speichern"
    ELSE
      IF t$="E"
        PRINT "Eingeben"
      ELSE
        PRINT "Unbekanntes Kommando"
      ENDIF
    ENDIF
  ENDIF
  '
LOOP
```

Die Verwendung von ELSE IF erzeugt ein kürzeres Listing mit weniger unübersichtlichen Einrückungen:

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="L"
    PRINT "Laden"
  ELSE IF t$="S"
    PRINT "Speichern"
  ELSE IF t$="E"
    PRINT "Eingeben"
  ELSE
    PRINT "Unbekanntes Kommando"
  ENDIF
  '
LOOP
```

Die Abarbeitung solcher Programmstrukturen erfolgt nach folgendem Muster:

Ist die Bedingung hinter IF erfüllt (hier t$="L"), so werden die Befehle zwischen IF und dem nächsten ELSE IF abgearbeitet (hier PRINT "Laden") und dann hinter den Befehl ENDIF verzweigt.

Ist eine der Bedingungen hinter den ELSE IF-Befehlen wahr, so werden alle Befehle bis zum nächsten ELSE, ELSE IF oder ENDIF (wenn kein ELSE existiert) abgearbeitet und anschließend zum Befehl hinter ENDIF gesprungen.

Ist weder die Bedingung hinter IF noch eine Bedingung hinter ELSE IF wahr, so werden die Befehle zwischen ELSE und ENDIF ausgeführt (sofern ein ELSE existiert).

Beispiele: Siehe oben.

Mehrfachverzweigungen

ON x **GOSUB** proc1,proc2, ...

```
x: iexp
proc1, proc2: Prozedurnamen, ohne Parameter
```

Dieser Befehl verzweigt zur x-ten Prozedur, die in der Liste hinter GOSUB aufgeführt ist. Dabei ist x ein numerischer Ausdruck, dessen Nachkommastellen (sofern vorhanden) ignoriert werden. Ist x kleiner als Eins oder größer als die Anzahl der Prozedurnamen hinter GOSUB, so wird keine Unterroutine aufgerufen. Nach Aufruf der Unterroutine wird das Programm hinter ON x GOSUB fortgesetzt.

Den Prozeduren können bei diesem Befehl keine Parameter übergeben werden.

Beispiel:

```
x=3
ON x GOSUB proc1,proc2,proc3
x=1
ON x+1 GOSUB proc1,proc2,proc3,proc4
```

--> Es wird zunächst die Prozedur proc3 aufgerufen, dann die Prozedur proc2.

SELECT x
CASE y [**TO** z] oder **CASE** y [,z,...]
CASE TO y
CASE y **TO**
DEFAULT
ENDSELECT
CONT

x,y,z: iexp oder String-Konstante mit einer Maximallänge von 4 Zeichen

Der Befehl SELECT ermöglicht Verzweigungen in Abhängigkeit vom Wert des numerischen Ausdrucks x. Folgendes Beispiel steht zur Erläuterung der dadurch entstehenden Programmstruktur:

```
x=0
SELECT x+2
CASE 1
  PRINT "x ist gleich 1"
CASE 2 TO 4
  PRINT "x ist gleich 2,3 oder 4"
CASE 5,6
  PRINT "x ist gleich 5 oder 6"
DEFAULT
  PRINT "x ist ungleich 1,2,3,4,5 oder 6"
ENDSELECT
```

--> Es wird 'x ist gleich 2,3 oder 4' auf den Monitor ausgegeben.

Zunächst wird der numerische Ausdruck hinter SELECT bestimmt, der die Verzweigungsbedingung ist (hier gleich 2). Anschließend werden die CASE-Anweisungen von oben nach unten durchgegangen und geprüft, ob hinter ihnen der aktuelle Wert der Verzweigungsbedingung zu finden ist. In diesem Beispiel steht hinter der ersten CASE-Anweisung nur die 1. Da die Verzweigungsbedingung gleich 2 ist, springt das Programm zum nächsten CASE.

Hinter dem zweiten CASE steht '2 TO 4'. Diese Bedingung ist also erfüllt, wenn das Verzweigungskriterium hinter SELECT Werte zwischen 2 und 4 (einschließlich) annimmt. Im vorliegenden Fall führt dies dazu, daß die Befehle zwischen dem zweiten und dem dritten CASE abgearbeitet werden. Danach wird die Programmausführung hinter dem Kommando ENDSELECT fortgesetzt.

Hinter dem dritten CASE ist eine weitere Variante der Angabe der möglichen Werte des SELECT-Kriteriums zu erkennen. Dort werden die gewünschten Werte in einer Liste, durch Kommata getrennt, aufgezählt.

Ist die aktuelle Ausprägung des Verzweigungskriteriums hinter keinem CASE aufgeführt, so werden die Befehle zwischen DEFAULT und ENDSELECT verarbeitet, sofern eine DEFAULT-Anweisung vorhanden ist. Statt DEFAULT kann aber auch OTHERWISE angegeben werden, was der Interpreter automatisch durch DEFAULT ersetzt.

Hinter CASE können jedoch nicht nur numerische Konstanten, sondern auch String-Konstanten mit einer maximalen Länge von vier Zeichen angegeben werden. Dabei kann der hinter CASE angegebene String maximal vier Zeichen enthalten. Bei Angabe nur eines Zeichens wird dessen ASCII-Wert als Verzweigungskriterium benutzt. Werden zwei Zeichen angegeben, so berechnet sich dieser Wert wie folgt:

ASCII-Wert des ersten + 255 * ASCII-Wert des zweiten Zeichens. Entsprechend wird bei Angabe von drei bzw. vier Zeichen verfahren.

Beispiel:

```
raus!=FALSE
REPEAT
  taste%=inp(2)
  SELECT taste%
  CASE "a" TO "z"
    PRINT "der Kleinbuchstabe "+chr$(taste%)+" wurde eingegeben"
  CASE "A" TO "Z"
    PRINT "der Großbuchstabe "+chr$(taste%)+" wurde eingegeben"
  CASE 27
    raus!=TRUE   !Programmende durch Drücken von <Esc>
  DEFAULT
    PRINT "eine unzulässige Taste wurde gedrückt!"
  ENDSELECT
UNTIL raus!
```

--> Innerhalb einer Schleife wird die Tastatur abgefragt und je nach gedrückter Taste verzweigt sowie das entsprechende Zeichen ausgegeben bzw. eine unzulässige Eingabe gemeldet. Als Abbruchkriterium für die Schleife dient das Drücken der Esc-Taste.

Das nächste Beispiel illustriert die Bedeutung der CONT-Anweisung, die nur dann eine Wirkung hat, wenn sie vor einem CASE- oder DEFAULT-Befehl steht. Dieser CONT-Befehl darf nicht mit dem gleichnamigen Kommando zur Wiederaufnahme eines unterbrochenen Programmlaufs verwechselt werden.

Beispiel:

```
x=1
SELECT x
CASE 1
  PRINT "x ist gleich 1"
  CONT
CASE 2
  PRINT "x ist gleich 2"
CASE 1,3
  PRINT "x ist gleich 3"
DEFAULT
  PRINT "x ist ungleich 1,2 und 3"
ENDSELECT
```

--> Es wird 'x ist gleich 1' und 'x ist gleich 2' auf dem Monitor.

Das CONT-Kommando sorgt dafür, daß die hinter ihm stehende CASE- oder DEFAULT-Anweisung übersprungen wird. In diesem Beispiel ist der Wert hinter der ersten CASE-Anweisung gleich der Verzweigungsbedingung. Folglich wird der Befehl PRINT "x ist gleich 1" ausgeführt. Anschließend folgt eine CONT-Anweisung, die die Zeile CASE 3 überspringt, obwohl die Verzweigungsbedingung nicht erfüllt, also ungleich 1, ist. Das führt dazu, daß auch der Befehl PRINT "x ist gleich 2" ausgeführt wird. Beim Erreichen der nächsten CASE- oder DEFAULT-Anweisung erfolgt nun der Sprung hinter ENDSELECT. Dies geschieht auch dann, wenn (wie hier) hinter diesem CASE die aktuelle Ausprägung der Verzweigungsbedingung aufgeführt ist.

Beispiel:

```
SELECT INP(2)
CASE "a" TO "g"
  PRINT "a bis g"
DEFAULT
  PRINT "default"
ENDSELECT
```

--> Wenn Sie eine der Tasten a,b,c,d,e,f oder g drücken, erscheint der Text 'a bis g', bei einer anderen Taste (nicht Tastaturumschalttaste) erscheint 'default'.

Bei CASE können die verschiedenen Möglichkeiten kombiniert werden, es ist also nicht nötig zu schreiben:

```
SELECT a$
CASE "a" TO "z"
  CONT
CASE "A" TO "Z"
  CONT
CASE "ä","ö","ü","ß","Ä","Ö","Ü"
  PRINT "OK"
ENDSELECT
```

sondern es reicht (auch):

```
SELECT a$
CASE "a" TO "Z", "A" TO "Z","ä","ö","ü",...
...
...
```

Schleifen

FOR, TO, STEP, NEXT
REPEAT, UNTIL
WHILE, WEND
DO, LOOP, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL
EXIT IF

GFA-BASIC 3.0 verfügt über eine ungewöhnlich große Auswahl an Schleifentypen. Normalerweise unterscheidet man abweisende und nicht abweisende Schleifen. Abweisende Schleifen sind solche, bei denen die Abbruchbedingung vor dem Eintritt in die Schleife geprüft wird, während diese Bedingung bei nicht abweisenden Schleifen am Ende des Schleifenkörpers überprüft wird. Die Konsequenz davon ist, daß nicht abweisende Schleifen mindestens einmal durchlaufen werden.

GFA-BASIC kennt zunächst einmal zwei der üblichen nicht abweisenden Schleifentypen, nämlich FOR-NEXT- und REPEAT-UNTIL-Schleifen. Ein dritter Standardschleifentyp ist die abweisende WHILE-WEND-Schleife. Als vierter Schleifentyp steht DO-LOOP zur Verfügung. Bei diesem Typ handelt es sich eigentlich nur um eine Endlosschleife ohne Abbruchbedingung, in GFA-BASIC kann dieser Befehl jedoch sehr variabel eingesetzt werden. Sowohl hinter DO als auch hinter LOOP können die Erweiterungen WHILE und UNTIL gesetzt werden, so daß sich z.B. auch Schleifen konstruieren lassen, die am Schleifenanfang und -ende eine logische Bedingung abfragen.

In jede der hier genannten Schleifentypen lassen sich darüber hinaus noch beliebig viele Abbruchbedingungen in den Schleifenkörper einbauen (EXIT IF).

FOR i = a **TO** e [**STEP** s]
(Befehle)
NEXT i
DOWNTO

```
i: avar
a,e,s: aexp
```

Die FOR-NEXT-Schleife (auch Zählschleife genannt) dient dazu, eine zwischen den Befehlsworten FOR und NEXT stehende Befehlsgruppe (Schleifenkörper) wiederholt abzuarbeiten. Zu diesem Zweck wird die Laufvariable i zunächst mit dem Anfangswert a versehen. Nun werden die Befehle des Schleifenkörpers durchlaufen, bis NEXT erreicht wird. Dort wird die Variable i um den Wert s erhöht, der hinter STEP steht. Ist kein STEP vorhanden, so wird i um den Wert eins erhöht. Jetzt wird geprüft, ob i den Wert e überschritten hat. Wenn dies der Fall ist, wird der Befehl hinter NEXT angesprungen, wenn nicht, so wird wieder beim ersten Befehl des Schleifenkörpers begonnen. Dieser Vorgang wird wiederholt, bis i größer e ist.

Eine Konsequenz dieses Ablaufs ist, daß i nach dem Verlassen der Schleife immer gleich dem ersten Wert ist, der das Abbruchkriterium überschritten hat. Der Inhalt der FOR-Schleife wird mindestens einmal durchlaufen.

Statt der STEP-Schrittweite s = -1 kann auch die Anweisung DOWNTO anstelle von TO verwendet werden. Im Zusammenhang mit DOWNTO ist die Verwendung von STEP dann nicht mehr möglich.

Als Laufvariable i sollte man Integervariablen verwenden, weil dadurch die Schleifenbefehle schneller abgearbeitet werden können als mit Fließkomma-Laufvariablen. Dies ist natürlich bei nicht ganzzahligen Schrittweiten nicht möglich.

Statt des Befehlswortes NEXT, gefolgt von der Laufvariable (z.B. i%), kann auch endfor i% geschrieben werden, was der Interpreter durch NEXT i% ersetzt.

Beispiele:

```
FOR i=1 TO 10
  PRINT i'
NEXT i
FOR i=-1 DOWNTO -10
  PRINT i'
NEXT i
```

--> Schreibt die Zahlen
1 2 3 4 5 6 7 8 9 10 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
auf den Bildschirm.

```
a$="T*e*s*t*w*o*r*t"
FOR j=1 TO LEN(a$) STEP 2
  PRINT MID$(a$,j,1);
NEXT j
```

--> Es erscheint 'Testwort' auf dem Bildschirm.

REPEAT
(Befehle)
UNTIL bed

bed: bexp

Mit REPEAT - UNTIL bed wird eine Anzahl von Befehlen eingerahmt, die ausgeführt werden, bis die logische Bedingung bed wahr ergibt.

Wird im Programm die Anweisung REPEAT erreicht, so werden die dahinter stehenden Befehle bis zum Erreichen von UNTIL abgearbeitet. Nun wird geprüft, ob die hinter UNTIL stehende logische Bedingung bed wahr ist. Wenn dies der Fall ist, so werden die Befehle hinter UNTIL ausgeführt. Ergibt bed aber falsch (0), dann springt die Programmausführung wieder zur REPEAT-Anweisung.

Die Befehle zwischen REPEAT und UNTIL werden mindestens einmal abgearbeitet, sofern sie nicht mit EXIT IF oder GOTO verlassen wird. Es handelt sich um eine nicht abweisende Schleife.

Statt UNTIL bed kann auch ENDREPEAT bed geschrieben werden, was der Interpreter automatisch durch UNTIL ersetzt.

Beispiele:

```
REPEAT
UNTIL MOUSEK
```

--> Wartet auf einen Maustastendruck.

```
i=1
REPEAT
  INC i
  j=SQR(i)
UNTIL i>10 and FRAC(j)=0
PRINT i
```

--> Es wird die Zahl 16 auf dem Bildschirm ausgegeben.

WHILE bed
(Befehle)
WEND

bed: bexp

Die Anweisungen WHILE und WEND können eine Gruppe von Befehlen einschließen, die solange abgearbeitet werden, wie die logische Bedingung bed wahr ist. Stößt GFA-BASIC auf den Befehl WHILE, so wird die dahinter stehende logische Bedingung geprüft. Wenn sie wahr ist, so werden die Befehle zwischen WHILE und WEND ausgeführt. Beim Erreichen von WEND springt das Programm wieder zu WHILE und der Zyklus beginnt von neuem, bis bed falsch wird. Statt der WEND-Anweisung kann auch ENDWHILE geschrieben werden, was der Interpreter durch WEND ersetzt.

Beispiel:

```
WHILE INKEY$=""
  PLOT MOUSEX,MOUSEY
WEND
```

--> Zeichnen mit der Maus, bis eine Taste gedrückt wird. Wenn schon ein Zeichen im Tastaturpuffer steht, so wird kein Punkt gesetzt.

DO
(Befehle)
LOOP

Die Befehle DO LOOP erzeugen eine Endlosschleife. Das Programm arbeitet die zwischen DO und LOOP stehenden Befehle ab und springt beim Erreichen von LOOP wieder zu dem Kommando DO zurück.

Die Schleife kann vom Programm nur mit Hilfe von EXIT IF, GOTO oder Programmende-Befehlen verlassen werden. Statt der LOOP-Anweisung kann auch ENDDO geschrieben werden, was der Interpreter durch LOOP ersetzt.

Beispiel:

```
DEFFILL 1,2,4
DO
  MOUSE mx,my,mk
  IF mk
    PBOX mx,my,mx+25,my+25
  ENDIF
LOOP
```

--> Zeichnet bei Maustastendruck ausgefüllte Rechtecke an der aktuellen Maus-Cursor-Position.

DO WHILE bed
DO UNTIL bed
LOOP WHILE bed
LOOP UNTIL bed

```
bed: bexp
```

Die Befehle DO und LOOP können um die Zusätze UNTIL und WHILE erweitert werden. Der Schleifenkopf DO WHILE führt dazu, daß das Innere der Schleife nur bearbeitet wird, solange bed wahr ist. Beginnt die Schleife mit DO UNTIL, so wird sie nur betreten, wenn die Bedingung bed nicht erfüllt ist.

LOOP WHILE führt dazu, daß das Programm zu DO zurückspringt, solange bed wahr ist. Für LOOP UNTIL gilt, daß bed falsch sein muß, damit der Rücksprung zum Schleifenanfang erfolgt.

Daher sind die Bedingungen bei DO abweisend, bei LOOP nicht abweisend.

```
DO WHILE bed                    WHILE bed
.                               .
.            entspricht         .
.                               .
LOOP                            WEND

DO                              REPEAT
.                               .
.                               .
.                               .
LOOP UNTIL bed                  UNTIL bed
```

Die Befehlsvarianten DO, DO WHILE und DO UNTIL können beliebig mit LOOP, LOOP WHILE und LOOP UNTIL kombiniert werden, so daß sich mit diesen Befehlen insgesamt neun Schleifenarten konstruieren lassen.

Beispiele:

```
DO
LOOP UNTIL MOUSEK
```

--> Wartet auf den Druck einer Maustaste.

```
DO UNTIL MOUSEK=2
  DO WHILE MOUSEK=1
    LINE 0,0,MOUSEX,MOUSEY
  LOOP
LOOP UNTIL INKEY$="a"
```

--> Zeichnet bei gedrückt gehaltener linker Maustaste Linien. Wenn die rechte Maustaste oder 'a' gedrückt wird, endet das Programm.

```
DO UNTIL EOF(#1)
  INPUT #1,a$
LOOP
```

--> Liest aus Kanal 1 sequentiell Zeichenketten, bis das Dateiende erreicht wird.

```
WHILE NOT EOF(#1)
  INPUT #1,a$
WEND
```

--> Konstruktion mit WHILE WEND ist langsamer, da zusätzlich NOT benötigt wird.

EXIT IF bed

```
bed: bexp
```

Mit Hilfe von EXIT IF kann aus einer Schleife gesprungen werden, wenn die bool'sche Bedingung bed erfüllt ist. Der Schleifentyp ist dabei beliebig wählbar.

Im Gegensatz zu älteren Versionen ist EXITIF auch innerhalb von IF-ENDIF und SELECT-ENDSELECT möglich.

Beispiel:

```
DO
  EXIT IF MOUSEK
LOOP
REPEAT
  EXIT IF INKEY$="x"
UNTIL FALSE
```

--> Beendet das Programm, wenn zunächst eine Maustaste und dann die Taste 'x' gedrückt wird.

Prozeduren und Funktionen

In GFA-BASIC 3.0 erhalten Unterprogramme, wie in jeder modernen Programmiersprache, Namen. Diesen Unterprogrammen können Parameter übergeben werden, sie können auch über VAR-Parameter auf verschiedene Variablen in verschiedenen Aufrufen angewandt werden.

Es kann sowohl der Wert einer Variablen als auch die Variable selbst übergeben werden (VAR). Im zweiten Fall kann die Übergabevariable von der Prozedur verändert werden. Es existieren also ein "call by value" und ein "call by reference".

Ebenso ist die Nutzung von lokalen Variablen möglich, d.h. es muß keine Rücksicht auf mögliche Namenskollisionen mit den aufrufenden Prozeduren bzw. Funktionen genommen werden.

Mit Hilfe von DEFFN kann man einzeilige Funktionen definieren, die später mit FN unter einem Namen angesprochen werden können. Es sind auch mehrzeilige Funktionen möglich (FUNCTION). Diese stellen grundsätzlich nur eine Sonderform der Prozeduren dar und liefern ein Ergebnis (über RETURN) zurück.

GOSUB proc [(par1,par2,...)]
PROCEDURE proc [(var1,var2,...)]
RETURN

```
proc: Name der Prozedur
par1,par2: sexp,aexp
var1,var2: svar,avar
```

Zwischen den Befehlsworten PROCEDURE und RETURN stehen die Befehle eines Unterprogramms. Hinter PROCEDURE steht der Name des Unterprogramms und eventuell die Liste der zu empfangenden Variablen. Der Aufruf einer PROCEDURE erfolgt unter Angabe ihres Namens am Zeilenanfang, gegebenenfalls gefolgt von entsprechenden Parametern, die in Klammern gesetzt sind. Zur Verdeutlichung, daß es sich dabei nicht um GFA-BASIC-Befehle handelt, kann wahlweise ein "@" oder das Schlüsselwort "GOSUB" vorangestellt werden. Bei Verwechslungsmöglichkeiten mit GFA-BASIC-Befehlen ist dies vorgeschrieben (z.B. @stop, @rem).

Die Übergabeparameter können Konstanten, Variablen und Ausdrücke sein. Von Variablen können nicht nur die Werte, sondern auch die Variablen selbst übergeben werden (siehe VAR).

Wird bei der Programmausführung die Anweisung RETURN erreicht, so springt das Programm zu dem Befehl, der hinter GOSUB steht. Statt des Befehlswortes PROCEDURE kann auch SUB geschrieben werden, statt RETURN auch ENDPROC oder ENDSUB. Dies ersetzt der Interpreter selbst.

Beispiel:

```
GOSUB slow_print("** Handbuch zu **")
@slow_print("* GFA-BASIC 3.0 *")
slow_print("GFA-SYSTEMTECHNIK")
'
PROCEDURE slow_print(t$)
  LOCAL i%
  FOR i%=1 TO LEN(t$)
    PRINT MID$(t$,i%,1);
    PAUSE 3
  NEXT i%
  PRINT
RETURN
```

--> Langsames, zeichenweises Ausgeben eines Strings.

```
a=8
@dritte_wurzel(a)
PRINT a
'
PROCEDURE dritte_wurzel(VAR x)
  x=x^(1/3)
RETURN
```

--> Berechnet die dritte Wurzel von 8 und gibt 2 auf dem Bildschirm aus.

LOCAL var1 [,var2,var3,...]

```
var1,var2,var3: avar,svar
```

Es ist mit Hilfe des Befehls LOCAL möglich, den Geltungsbereich von Variablen einzuschränken. Die hinter LOCAL aufgeführten Variablen sind nur in der Prozedur gültig, in der LOCAL steht sowie in allen von dieser Prozedur aufgerufenen Unterprogrammen.

Hinter LOCAL dürfen auch Variablen stehen, die im Hauptprogramm, also global, gültig sind. Diese Variablen können dann in der Unterroutine nicht angesprochen werden, stehen nach dem Verlassen der Prozedur aber wieder unverändert zur Verfügung.

Übergabevariablen einer Prozedur oder Funktion sind immer lokal.

Beispiel:

```
x=2
GOSUB test
PRINT x,y
'
PROCEDURE test
  LOCAL x,y
  x=3
  y=4
RETURN
```

--> Auf dem Bildschirm erscheinen die Zahlen 2 und 0.

@funk [(par1,par2,...)]
FUNCTION funk [(var1,var2,...)]
RETURN exp
ENDFUNC

```
funk: Name der Funktion
par1,par2: sexp,aexp
var1,var2: svar,avar
exp: sexp, aexp
```

Zwischen den Befehlsworten FUNCTION und ENDFUNC stehen die Befehle einer Unterroutine (ähnlich PROCEDURE). Dabei stehen hinter FUNCTION der Name der Unterroutine und eventuell die Liste der zu übergebenden Variablen. Der Aufruf der Unterroutine erfolgt durch die Angabe des "Klammeraffen" @ oder FN und des Funktionsnamens. Dann folgt gegebenenfalls eine Parameterliste.

Die Übergabeparameter können Konstanten, Variablen und Ausdrücke sein. Von Variablen können nicht nur die Werte, sondern auch die Variablen selbst übergeben werden (siehe VAR).

Wird bei der Programmausführung die Anweisung RETURN erreicht, so wird der in der Funktion ermittelte Wert an der Stelle des Funktionsaufrufes eingesetzt.

In einer Funktion kann RETURN auch mehrfach verwendet werden, mit IF o.ä., es ist aber nicht erlaubt, eine Funktion ohne RETURN zu beenden (durch ENDFUNC).

Ein $-Zeichen am Ende eines Funktionsnamens kennzeichnet Funktionen mit String-Ergebnis.

Beispiel:

```
fl%=@fak_loop(15)
fr%=@fak_rekurs(10)
'
PRINT "Schleife:  fak(15) = ";fl%
PRINT "Rekursion: fak(10) = ";fr%
'
FUNCTION fak_loop(f%)
  w=1
  FOR j%=1 TO f%
    MUL w,j%
  NEXT j%
  RETURN w
ENDFUNC
'
FUNCTION fak_rekurs(f%)
  IF f%<2
    RETURN 1
  ELSE
    RETURN f%*@fak_rekurs(PRED(f%))
  ENDIF
ENDFUNC
```

--> Die Fakultäten von 10 und 15 werden innerhalb einer Schleife und einmal rekursiv berechnet.

DEFFN func [(x1,x2,...)] = ausd
FN func [(y1,y2,...)]

```
func,x1,x2: var
ausd,y1,y2: exp
```

Der Befehl DEFFN erlaubt das Definieren von einzeiligen Funktionen. Die Definition dieser Funktion steht im Ausdruck ausd, der sowohl einen numerischen als auch einen String-Ausdruck ergeben kann. Die Funktion wird nach ihrer Definition mit FN func oder dem Klammeraffen @ aufgerufen.

Die in der Funktionsdefinition enthaltenen Variablen werden als Parameter übergeben. In der Funktionsdefinition notiert man zu diesem Zweck die Variablennamen x1, x2,... die in ausd auftauchen. Beim Aufruf der Funktion können die Parameter y1, y2,... als numerischer oder String-Ausdruck übergeben werden.

Wenn die Variablen x1, x2,... auch global definiert sind, so können sie im Ausdruck ausd nicht angesprochen werden, da sie innerhalb von ausd die übergebenen Werte enthalten. Die Variablen x1, x2,... sind also für die Funktion lokale Variablen.

Die Anweisung FN kann auch durch @ ersetzt werden. Funktionen können beliebig verschachtelt werden, bei DEFFN allerdings nicht rekursiv (Abbruchbedingung nicht möglich).

Beispiele:

```
DEFFN test(y,a$)=x-y+LEN(a$)
x=2
PRINT @test(4,"abcdef")
```

--> Es erscheint die Zahl 4 (2-4+6) auf dem Bildschirm.

```
DEFFN first_last$(a$)=LEFT$(a$)+RIGHT$(a$)
b$=@first_last$("TEST")
PRINT b$
```

--> Es erscheint der Text "TT" auf dem Monitor.

```
DEFFN hoch_vier(x)=x^4
DEFFN vierte_wurzel(x)=x^(1/4)
PRINT @vierte_wurzel(@hoch_vier(1024))
```

--> Es erscheint die Zahl 1024.

Ereignisbezogene Verzweigungen

In diesem Abschnitt werden zwei Arten von Ereignissen behandelt, deren Abfrage in GFA-BASIC möglich ist. Nicht GFA-BASIC spezifische Ereignisse, wie z.B. die Abfrage von Mausklicks, Auswahl aus Pull-Down-Menüs usw., werden in anderen Kapiteln behandelt (Menü- und Fensterprogrammierung, AES-Bibliotheken).

Das erste der zwei besprochenen Ereignisse ist das gleichzeitige Drücken der Tasten Control, Shift und Alternate. Die zweite Ereignisart ist das Auftreten eines Fehlers.

ON BREAK
ON BREAK CONT
ON BREAK GOSUB proc

proc: Name einer Prozedur

Diese drei Befehle legen die Reaktion auf das gleichzeitige Drücken der Tasten Control, Shift (nur linke Shift-Taste) und Alternate fest. Drücken dieser Tastenkombination führt gewöhnlich zum Programmabbruch, kann aber auch zum Anspringen einer bestimmten Prozedur dienen. Zu diesem Zweck legt man mit ON BREAK GOSUB die anzuspringende Prozedur proc fest.

Mit ON BREAK CONT erreicht man, daß auf das Drücken der genannten Tastenkombination nicht mehr reagiert wird. ON BREAK schaltet die normale Reaktion (Programmabbruch) wieder ein.

Beispiel:

```
ON BREAK GOSUB test
PRINT "Drücken Sie CONTROL, SHIFT (links) und ALTERNATE"
DO
LOOP
'
PROCEDURE test
  PRINT "Das war's"
  ON BREAK
RETURN
```

--> Es erscheint die Aufforderung zum Drücken der Tastenkombination. Wird sie gedrückt, schaltet die Prozedur test die normale Break-Routine wieder ein.

ON ERROR
ON ERROR GOSUB proc
RESUME [NEXT]
RESUME [mar]

```
proc: Name einer Prozedur
mar: Name einer Marke
```

Das Auftreten eines Fehlers (TOS- oder GFA-BASIC-spezifischer Fehler) führt normalerweise zur Ausgabe einer Fehlermeldung und zum Programmabbruch. Mit ON ERROR GOSUB hat man die Möglichkeit, beim Auftreten eines Fehlers zur Prozedur proc zu verzweigen. In dieser Prozedur kann man die Reaktion auf Fehler festlegen.

Mit dem Befehl ON ERROR schaltet man wieder auf die normale Fehlerabfangung zurück, also auf Ausgabe einer Fehlermeldung und Programmabbruch. Wenn ein Fehler auftritt, so wird automatisch ein ON ERROR-Befehl ausgeführt. Um auf mehrere hintereinanderfolgende Fehler reagieren zu können, muß in der Fehlerabfangroutine somit der Befehl ON ERROR GOSUB proc noch einmal enthalten sein.

Der RESUME-Befehl erlaubt eine spezielle Reaktionsweise auf aufgetretene Fehler; er ist nur in der Fehlerabfangprozedur sinnvoll. Mit RESUME NEXT kann erreicht werden, daß der nächste Befehl angesprungen wird, der hinter der Programmanweisung steht, die zum Fehler führte. RESUME mar führt dazu, daß die Marke mar angesprungen wird. Der Befehl RESUME ohne Angabe von NEXT oder einer Marke springt den Befehl, bei dem der Fehler auftrat, erneut an. Trat ein fataler Fehler auf (siehe FATAL), so kann nur der Befehl RESUME mar benutzt werden, nicht aber RESUME NEXT oder RESUME ohne Markenangabe.

Beispiel:

```
ON ERROR GOSUB fehlerabfanger
ERROR 5
PRINT "und nochmal ..."
ERROR 5
PRINT "wird nicht erreicht"
'
PROCEDURE fehlerabfanger
  PRINT "Ok, Fehler abgefangen."
  RESUME NEXT
RETURN
```

--> Es erscheinen die Texte 'Ok, Fehler abgefangen.' und 'und nochmal ...' auf dem Bildschirm; dann die durch ERROR 5 ausgelöste Fehlermeldung 'Quadratwurzel nur für positive Zahlen'. Die Marke für RESUME darf sowohl in einer Prozedur als auch im Hauptprogramm stehen.

ERROR x
ERR
ERR$(x)
FATAL

x: aexp

Mit ERROR läßt sich der Fehler mit der Nummer x auslösen (Fehlermeldungstabelle siehe Anhang). Dies ist z.B. beim Austesten einer Fehlerbehandlungsroutine sinnvoll.

In der Variable ERR steht die Nummer des aufgetretenen Fehlers. Mit ihrer Hilfe kann man in einer Fehlerbehandlungsroutine bestimmten Fehlern spezifische Reaktionen zuordnen.

Die Funktion ERR$ liefert den String der GFA-BASIC-Fehlermeldung mit der Nummer x.

Die Variable FATAL ist wahr, wenn der aufgetretene Fehler an einer dem Interpreter nicht bekannten Adresse auftauchte. Dies kann z.B. dann der Fall sein, wenn der Fehler bei der Abarbeitung einer Betriebssystemroutine auftrat. Die Konsequenz für das Programm besteht darin, daß nach einem fatalen Fehler ein RESUME oder RESUME NEXT nicht mehr korrekt bearbeitet werden kann.

Beispiele:

```
ON ERROR GOSUB fehlerabfangung
INPUT "Welchen Fehler hätten Sie denn gern: ",e
ERROR e
'
PROCEDURE fehlerabfangung
  PRINT "Dies war Fehler-Nr.: ";ERR
  IF FATAL
    PRINT "der ein fataler Fehler ist."
  ENDIF
RETURN
```

--> Fragt den Benutzer nach der gewünschten Fehlernummer und gibt diese Nummer wieder aus.

```
~FORM_ALERT(1,ERR$(100))
```

--> Gibt die Fehlermeldung mit der Kennung 100 (d.h. die Copyright-Meldung) als Alert-Box aus.

Interrupt-Programmierung

EVERY ticks **GOSUB** proc
EVERY STOP
EVERY CONT
AFTER ticks **GOSUB** proc
AFTER STOP
AFTER CONT

```
ticks: iexp
proc: Name einer Prozedur
```

Mit Hilfe der Befehle EVERY und AFTER können Prozeduren nach Ablauf einer bestimmten Zeit ticks aufgerufen werden. Der Befehl EVERY führt dazu, daß die Prozedur proc alle ticks Zeiteinheiten aufgerufen wird; AFTER verursacht, daß diese Prozedur einmal nach Ablauf der ticks Zeiteinheiten bearbeitet wird.

Die Zeiteinheit wird dabei in zweihundertstel Sekunden angegeben (ticks = 200 bedeutet also eine Sekunde). Allerdings kann die Verzweigung zu der angegebenen Prozedur nur bei jeder vierten Zeiteinheit durchgeführt werden, so daß die effektive zeitliche Auflösung nur eine fünfzigstel Sekunde beträgt.

Mit Hilfe von EVERY STOP kann das Anspringen der Prozedur nach Ablauf der Zeitfrist abgeschaltet werden, mit EVERY CONT wird es wieder fortgesetzt. Die Befehle AFTER STOP und AFTER CONT arbeiten analog. GFA-BASIC-intern werden diese Befehle über den etv_timer Vektor ($400) realisiert.

Es wird jeweils nur nach vollständiger Abarbeitung eines Befehls geprüft, ob eine solche Prozedur ausgeführt werden soll, d.h. langsam ablaufende Befehle wie INP(2), QSORT, Dateioperationen oder ähnliches können diese Routinen behindern.

Beispiel:

```
EVERY 4 GOSUB lines
linien!=TRUE
GRAPHMODE 3
DEFFILL 1,0
PLOT MOUSEX,MOUSEY
REPEAT
  IF MOUSEK=1
    EVERY STOP
  ELSE
    EVERY  CONT
  ENDIF
  DRAW TO MOUSEX,MOUSEY
UNTIL MOUSEK=2
'
PROCEDURE lines
  INC y%
  LINE 320,y%,639,y%
  IF y%=399
    y%=0
  ENDIF
RETURN
```

--> Läßt Linien in der rechten Bildschirmhälfte von oben nach unten wandern und erlaubt gleichzeitig das Zeichnen mit der Maus. Das Drücken der linken Maustaste schaltet die wandernden Linien ein bzw. aus. Das Programm kann durch das Drücken der rechten Maustaste abgebrochen werden.

```
PRINT "In 3 Sekunden folgt ein Text,"
PRINT "wenn Sie keine Taste drücken."
AFTER 600 GOSUB text
REPEAT
UNTIL INKEY$<>"" OR RAUS!
AFTER STOP
'
PROCEDURE text
  PRINT
  PRINT "Hier ist der Text"
  raus!=TRUE
RETURN
```

--> Wird in den drei Sekunden nach dem Programmstart keine Taste gedrückt, so erscheint ein Text. Wird eine Taste gedrückt, so beendet dies das Programm.

Sonstige

REM, GOTO, PAUSE, DELAY
END, EDIT
STOP
SYSTEM, QUIT

REM x
' x
<Befehlszeile> ! x

x: beliebiger Text

In einer Zeile, die mit REM oder ' beginnt, können beliebige Texte stehen. Diese Texte unterliegen nicht der Syntaxkontrolle des Editors und werden bei der Programmausführung nicht beachtet. Außerdem kann an das Ende einer Befehlszeile hinter dem Ausrufezeichen ! ein beliebiger Kommentar angefügt werden. Im Zusammenhang mit DATA-Anweisungen ist die Benutzung dieses Zeichens allerdings nicht möglich. Auch bei INLINE ist kein Kommentar möglich.

Beispiel:

```
REM Kommentar
' PRINT "Kommentar"
PRINT "REM"  ! Kommentar
```

--> Auf dem Bildschirm erscheint das Wort REM.

GOTO mar
mar:

```
mar: vom Programmierer definierte Marke
```

Mit Hilfe einer Marke mar: kann man Stellen im Programm festlegen, die mit dem Befehl GOTO angesprungen werden können. Die Programmausführung wird dann an der Markenposition fortgesetzt. Die Marke kann aus Buchstaben, Ziffern, Unterstrichen und Punkten bestehen. Sie darf auch, im Unterschied zu Variablennamen, mit einer Ziffer beginnen, muß aber mit einem Doppelpunkt enden. Beim Ansprung der Marke mit GOTO wird der Doppelpunkt allerdings nicht angegeben.

Es ist mit Hilfe des GOTO-Befehls nicht möglich, in Prozeduren oder Funktionen hinein- oder aus ihnen herauszuspringen. Dies gilt auch für FOR-NEXT-Schleifen. GOTO-Befehle machen Programme meist sehr schnell unübersichtlich, weshalb man sie nach Möglichkeit vermeiden sollte.

Beispiel:

```
PRINT "Stelle 1"
GOTO sprungmarke
PRINT "Stelle 2"
sprungmarke:
PRINT "Stelle 3"
```

--> Auf dem Bildschirm erscheinen die Texte "Stelle 1" und "Stelle 3".

PAUSE x
DELAY x

x: aexp

Der Befehl PAUSE hält die Programmausführung für x / 50 Sekunden an. DELAY hat eine ähnliche Wirkung, jedoch wird die Ausführung des Programms für x Sekunden unterbrochen (theoretische Auflösung in Millisekunden). DELAY benutzt die GEM-Routine EVNT_TIMER, ist also für GEM-Programme zu empfehlen.

Beispiel:

```
PRINT "Start"
PAUSE 100
PRINT "eine Pause"
DELAY 2
PRINT "Ende"
```

--> Es erscheinen der Text 'Start' und zwei Sekunden später 'eine Pause', nach weiteren 2 Sekunden 'Ende'.

END
EDIT
STOP

Diese Befehle dienen der Beendigung des Programms. Der Befehl END hält die Programmausführung an und läßt eine Alert-Box mit dem Text 'Programmende' erscheinen. Nach dem Anwählen der einzigen Wahlalternative kehrt GFA-BASIC 3.0 in den Editor zurück.

EDIT beendet die Programmausführung und kehrt sofort in den Editor zurück.

STOP läßt eine Alert-Box mit den Wahlalternativen STOP und CONT erscheinen. Durch die Wahl von CONT kann man die Programmausführung fortsetzen. Nach der Wahl von STOP geht GFA-BASIC in den Direktmodus. Dort kann man nun die Werte von Variablen verändern oder abfragen und durch CONT die Programmausführung wieder fortsetzen.

Beispiel:

```
x=3
STOP
PRINT x
```

--> Wählen Sie den Button 'STOP', wenn die entsprechende Alert-Box erscheint. Geben Sie nun im Direktmodus folgende Befehle ein:

```
PRINT x
```

--> Es erscheint die Zahl 3. Geben Sie ein:

```
x=4
CONT
```

--> Der letzte Befehl des Listings (PRINT x) wird nun abgearbeitet. Es erscheint die Zahl 4, also der im Direktmodus angegebene Wert.

NEW

Diese Anweisung löscht ein Programm. Im Direktmodus erfolgt eine Sicherheitsabfrage. Im Editor ist dieser Befehl durch Shift-F4 oder Mausclick (mit Sicherheitsabfrage) zu erreichen.

LOAD f$

```
f$: sexp
```

Der Befehl LOAD dient zum Laden eines GFA-BASIC-Programms. Der String-Ausdruck f$ enthält den Zugriffspfad der gewünschten Datei. Ohne Angabe einer Extension wird voreingestellt .GFA benutzt.

Beispiel:

```
LOAD "A:\TEST.GFA"
```

--> Lädt die Programmdatei TEST.GFA aus dem Wurzelverzeichnis von Laufwerk A.

SAVE f$
PSAVE f$

```
f$: sexp
```

Der Befehl SAVE speichert eine Programmdatei unter dem in f$ angegebenen Namen ab. Bei Verwendung des Befehls PSAVE wird die angegebene Datei mit List-Schutz abgespeichert (d.h. die Datei kann nach erneutem Laden mit LOAD nicht gelistet werden, sondern wird sofort ausgeführt). Ohne Angabe einer Extension wird .GFA verwendet.

Beispiel:

```
SAVE "A:\TEST.GFA"
```

--> Speichert die aktuelle Programmdatei unter dem Namen TEST.GFA auf Laufwerk A ab.

LIST [f$]
LLIST [f$]

```
f$: sexp
```

Der Befehl LIST zeigt das aktuelle Programm auf dem Bildschirm an. Optional kann ein Zugriffspfad angegeben werden, unter dem die Programmdatei im ASCII-Format abgespeichert werden kann. Programmdateien, die mit MERGE in andere Programme eingefügt werden sollen, müssen mit LIST oder SAVE,A (aus der Editor-Menüleiste) im ASCII-Format gespeichert sein.

Ohne Angabe einer Extension wird voreingestellt .LST verwendet. Mit LLIST kann das aktuelle Programm auf den Drucker ausgegeben werden.

Das Drucker-Listing kann nur durch Ausschalten des Drucker unterbrochen werden. Danach dauert es noch ca. 30 Sekunden, bis das Programm weiter ausgeführt bzw. in den Editor zurückgekehrt wird. (vgl. LLIST und die Punktbefehle im Abschnitt über den Editor). Es ist aber auch möglich die Ausgabe umzulenken. (vgl. OPEN)

Beispiel:

```
LIST "A:\TEST.LST"
```

--> Speichert das aktuelle Programm unter dem Namen "TEST.LST" im ASCII-Format auf Laufwerk A ab.

```
.ll 70
.pl 66
LLIST
```

--> Gibt das aktuelle Programm mit einer Zeilenlänge von 70 Zeichen und 66 Zeilen pro Seite auf den Drucker aus.

CHAIN f$

```
f$: sexp
```

Der Befehl CHAIN ermöglicht das Laden eines GFA-BASIC-Programms in den Arbeitsspeicher und startet das Programm. Ohne Angabe einer Extension wird voreingestellt .GFA verwendet.

Beispiel:

```
CHAIN "A:\ZUSATZ.GFA"
```

--> Lädt die Programmdatei ZUSATZ.GFA und startet das Programm.

RUN [f$]

```
f$ : sexp
```

Der Befehl RUN startet das aktuelle Programm. Wird zusätzlich ein vollständiger Dateiname angegeben, so wird das entsprechende Programm geladen und gestartet.

Beispiel:

```
RUN "A:\TEIL_2.GFA"
```

--> Lädt und startet das Programm mit dem Namen TEIL_2.GFA von Laufwerk A.

SYSTEM [n]
QUIT [n]

n: iexp

Die Befehle SYSTEM und QUIT sind in der Wirkung gleichwertig. Sie beenden den Programmlauf und verlassen GFA-BASIC. Im Unterschied zu den 2.xx Versionen liefern SYSTEM und QUIT optional einen 2-Byte-Integer-Wert zurück. Dieses Word ist Null, wenn der Interpreter korrekt verlassen wurde und wird an das aufrufende Programm (im Normalfall das Desktop) übergeben.

Dabei gilt die Konvention, daß Null einen fehlerfreien Durchlauf signalisiert, eine positive 16-Bit-Zahl kennzeichnet einen internen Fehler oder eine Warnung und eine negative 16-Bit-Zahl steht meist für die entsprechende Betriebssystem-Fehlermeldung. Dies wird aber nicht von allen Programmen eingehalten.

Beispiel:

```
RESERVE 100
PRINT EXEC(0,"GFABASIC.PRG","","")
```

--> Dieses kleine Programm starten (Shift-F10), dann wird nochmals ein GFA-BASIC geladen. In diesem GFA-BASIC den Befehl:

```
QUIT 23
```

--> eingeben und starten. Ausgabe: 23 und Programm-Ende des zuerst geladenen GFA-BASIC.

Fehlerbehandlung

TRON
TRON #n
TROFF

```
n: iexp
```

Der Befehl TRON (trace on) führt dazu, daß jeder ausgeführte Befehl am Bildschirm aufgelistet wird. Diese Liste kann unter Angabe einer Kanalnummer in eine Datei, auf den Drucker oder auf die serielle Schnittstelle usw. umgeleitet werden. Der Befehl TROFF schaltet dies wieder ab.

Beispiel:

```
PRINT "Start:"
TRON
FOR i%=1 TO 5
  PRINT i%
NEXT i%
TROFF
PRINT "Ende."
```

--> Es erscheint das Wort 'Start:'. Dann die Zahlen von 1 bis 5 und die Befehle, die zu deren Darstellung führen. Danach das Wort 'Ende'.

```
OPEN "O",#1,"\tron.lst"
TRON #1
FOR i%=1 TO 10
  PRINT i%
NEXT i%
TROFF
CLOSE #1
'
OPEN "O",#2,"prn:"
TRON #2
FOR i%=10 TO 630 STEP 10
  LINE i%,0,i%,100
NEXT i%
TROFF
CLOSE #2
```

--> Es erscheinen die Zahlen von 1 bis 10 sowie die dazu notwendigen Befehle und eine Reihe von vertikalen Linien im Abstand von 10 Pixeln. Diese Befehlsausgabe erfolgt auf Diskette bzw. Drucker.

TRON proc
TRACE$

```
proc: Name einer Prozedur
```

Mit Hilfe des Befehls TRON proc kann eine Prozedur angegeben werden, die vor der Bearbeitung eines jeden Befehls aufgerufen wird. Die Variable TRACE$ enthält den Befehl, der als nächster abgearbeitet wird. Die Anweisung TRON proc ermöglicht in Zusammenarbeit mit TRACE$ eine sehr effiziente Fehlersuche. So kann z.B. in Abhängigkeit vom nächsten zu verarbeitenden Befehl eine Ausgabe von bestimmten Variablen auf dem Drucker erfolgen, so daß die Veränderungen einer Variablen während des Programmlaufs verfolgt werden können.

Wichtig ist, daß die TRON-Przedur möglichst wenig stören sollte. Also keine PRINT-Befehle in Bildschirmmasken (TEXT, ATEXT, ...) oder auch keine Benutzung von VDI-Routinen (wegen GDOS-Benutzung oder DEFTEXT-LINE-A). Oder auch ...

Beispiel:

```
TRON tr_proc
GRAPHMODE 3
DO UNTIL MOUSEK
  x1%=100+RAND(200)
  y1%=100+RAND(100)
  x2%=200+RAND(200)
  y2%=200+RAND(100)
  PBOX x1%,y1%,x2%,y2%
LOOP
'
PROCEDURE tr_proc
  IF BIOS(11,-1) AND 4   ! Control-Taste
    adr%=XBIOS(2)
    BMOVE adr%+1280,adr%,4*1280
    PRINT AT(1,5);SPACE$(80);
    PRINT AT(1,5);LEFT$(TRACE$,79);
    PAUSE 20
  ENDIF
RETURN
```

--> Dieses Programm zeichnet zufällig verteilte Rechtecke auf den Bildschirm. Das Drücken der Control-Taste läßt die gerade abgearbeiteten Befehle auf dem Bildschirm erscheinen. Durch das Drücken einer Maustaste wird das Programm beendet.

DUMP [a$ [**TO** b$]]

```
a$,b$: sexp
```

Mit Hilfe des DUMP-Befehls können Variableninhalte während des Programmlaufs ausgegeben oder Labels, Prozeduren und Funktionen aufgelistet werden. Dazu kann der String-Ausdruck a$ folgende Werte annehmen:

Beispiele:

```
DUMP
```

--> Gibt alle Variablenwerte und die Dimensionierungen von Arrays aus.

```
DUMP "a"
```

--> Wie oben, aber nur Variablen bzw. Arrays, die mit 'a' beginnnen.

```
DUMP ":"
```

--> Listet alle Labels (Marken) und gibt dahinter die Editor-Zeilennummer an, wo die Marke definiert wird. Die Variation (:b) listet nur Marken, die mit 'b' beginnen.

```
DUMP "@"
```

--> Listet alle Prozeduren und Funktionen auf, wobei hinter den Namen die Zeilennummer steht, an der diese im Editor zu finden sind:

```
proc_name @ 100      (Prozedur)
func_name FN 200     (Funktion mit numerischer Rückgabe)
func_name $ FN 300   (Funktion mit String-Rückgabe)
```

Nicht mehr definierte Labels, Prozeduren und Funktionen werden ohne Zeilennummer angegeben. Wird das Programm-Listing unter der Option SAVE,A abgespeichert und neu eingeladen, erscheinen diese undefinierten Namen nicht mehr. Es erscheinen aber Label-, Prozedur- und Funktionsnamen ohne Zeilennummern, die verwendet werden, aber nicht definiert sind.

Zu den hinter den Namen angegebenen Zeilennummern kann im Editor mit Control + G gesprungen werden.

Bei der Ausgabe von Strings werden maximal 60 Zeichen dargestellt. Ist die Zeichenkette länger, wird als letztes Zeichen ein '>' ausgegeben, sonst ein ' " '. Steuerzeichen (d.h. ASCII-Wert < 32) werden durch '.' dargestellt.

Die oben genannten Ausgaben können auch in eine Datei umgeleitet werden. Dazu muß in b$ ein Dateiname angegeben werden.

Als Default-Extension (ohne .) wird .DMP angenommen.

8 - Grafik

Der ATARI ST kennt drei verschiedene Grafikmodi, einen Schwarz-Weiß-Modus und zwei Farb-Modi. Die Koordinaten der Grafikbefehle und die mit ihnen darstellbaren Farben sind von der aktuellen Auflösung abhängig, daher folgt hier nun eine Übersicht der Grafikmodi:

	Bildschirmpunkte (Koordinaten)	**Farben (Farbregister)**
Niedrige Auflösung:	320 x 200 (0 bis 319) (0 bis 199)	16 (0 bis 15) aus 512 Farben
Mittlere Auflösung:	640 x 200 (0 bis 639) (0 bis 199)	4 (0 bis 3) aus 16 Farben
Hohe Auflösung:	640 x 400 (0 bis 639) (0 bis 399)	2 (0 und 1)

Im ersten Abschnitt werden die Befehle für die Auswahl von Farben (SETCOLOR, COLOR) vorgestellt. Danach werden die Definitionsbefehle für die Auswahl und Erzeugung von verschiedenen Maus-Cursor-Formen, Markierungen, Füllmustern, Umrahmungen und Linientypen aufgeführt (DEFMOUSE, DEFMARK, DEFFILL, BOUNDARY, DEFLINE).

Der nächste Abschnitt stellt die CLIP-Befehle zur Begrenzung von Grafikausgaben und die allgemeinen Grafikbefehle für das Zeichnen verschiedener geometrischer Grundformen vor (PLOT, LINE, BOX, CIRCLE, ELLIPSE). Es werden Möglichkeiten zur Darstellung von Polygonen (POLYLINE, POLYMARK, POLYFILL) und Grafiktext (TEXT) beschrieben. Der Abschnitt endet mit der Vorstellung des FILL-Befehls.

Im letzten Abschnitt dieses Kapitels geht es um die Behandlung von Bildschirmausschnitten mit SGET, SPUT, GET und PUT.

Definitionsbefehle

SETCOLOR register,rot,gruen,blau
SETCOLOR register,mischwert
COLOR farbe

```
register,rot,gruen,blau,mischwert,farbe: iexp
```

Die erste Variante von SETCOLOR bestimmt den Anteil der Farben Rot, Grün und Blau in einem bestimmten Farbregister. Dabei wird die Intensität der Farbanteile durch eine Skala von 0 (gering) bis 7 (groß) festgelegt. Die Anzahl der verfügbaren Farben ist abhängig von der aktuellen Auflösung. In der zweiten Variante wird die Farbeinstellung über folgende Formel berechnet: mischwert = rot * 256 + gruen * 16 + blau, wobei die Werte für rot, gruen und blau wieder aus der Skala von 0 bis 7 stammen.

Das Setzen von Farbanteilen in den Farbregistern ist natürlich nur in den Farbauflösungen sinnvoll. Für die monochrome Auflösung gilt, daß dann, wenn für mischwert ein anderer Wert als 0 oder 1 angegeben wird, gerade Zahlen die gleiche Wirkung wie 0 und ungerade Zahlen die gleiche Wirkung wie 1 haben.

Der Befehl COLOR bestimmt die Zeichenfarbe. Dabei steht im numerischen Ausdruck farbe ein von der jeweiligen Auflösung abhängiger Wert zwischen 0 und 15.

Beispiel:

```
SETCOLOR 0,0
```

--> Auf dem Monochrommonitor erscheint weiße Schrift auf schwarzem Grund.

DEFMOUSE symbol
DEFMOUSE bitmuster$

```
symbol: iexp
bitmuster$: sexp
```

Die erste Befehlsvariante bestimmt das Maussymbol aus einer von 8 vordefinierten Formen. Dabei gilt folgende Zuordnung:

```
0 --> Pfeil
1 --> Doppelklammer
2 --> Biene
3 --> Zeigende Hand
4 --> Offene Hand
5 --> Dünnes Fadenkreuz
6 --> Fettes Fadenkreuz
7 --> Umrandetes Fadenkreuz
```

Die zweite Variante dient der Definition eines eigenen Maus-Cursors. Dazu müssen der Aktionspunkt, die Maskenfarbe, die Cursor-Farbe und die Bit-Muster für das Aussehen der Maske und das Aussehen des Maus-Cursors in einer Zeichenkette übergeben werden. Der Aktionspunkt ist derjenige Punkt des Maus-Cursors, dessen Koordinaten als Mausposition verwaltet werden. Wird die Mausposition abgefragt, so sind es die Koordinaten des Aktionspunktes, die zugerückgemeldet werden.

Dabei muß beachtet werden, daß alle diese Werte in Wortgröße übergeben werden müssen. Zu diesem Zweck kann man den Befehl MKI$ verwenden, so daß sich bitmuster$ wie folgt ergibt:

```
bitmuster$=MKI$(x-Koordinate Aktionspunkt)
          +MKI$(y-Koordinate Aktionspunkt)
          +MKI$(1)        ! normal, -1=XOR
          +MKI$(Maskenfarbe)
          +MKI$(Cursor-Farbe)
          +maske$  (Bit-Muster der Maske)
          +cursor$ (Bit-Muster des Cursors)
```

maske$ und cursor$ bestehen aus je 16 Worten, welche jeweils das Bit-Muster einer Zeile ergeben.

Beispiel:

```
DEFMOUSE 2
REPEAT
UNTIL MOUSEK
m$=MKI$(0)+MKI$(0)+MKI$(1)+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  m$=m$+MKI$(65535)
NEXT i%
FOR i%=1 TO 16
  m$=m$+MKI$(1)
NEXT i%
PBOX 0,0,200,100
DEFMOUSE m$
REPEAT
UNTIL MOUSEK
```

--> Es erscheint zunächst eine Biene als Maus-Cursor. Nach dem Drücken einer Maustaste hat der Maus-Cursor die Form eines Striches. Auf dem schwarzen Hintergrund links oben erscheint die Form eines Rechtecks, was durch die Definition der Maske verursacht wird. Die Warteschleife wird durch Drücken einer Maustaste verlassen.

DEFMARK [farbe] , [art] , [groesse]

```
farbe,art,groesse: iexp
```

DEFMARK legt Farbe, Art und Größe der Eckpunkte fest, die mit dem Befehl POLYMARK gesetzt werden. Für den numerischen Ausdruck farbe gelten je nach Auflösung Werte zwischen 0 und 15 (siehe die Übersicht dieses Kapitels). Mit art sind folgende Markierungen möglich:

```
1 --> Punkt
2 --> Plus-Zeichen
3 --> Stern
4 --> Rechteck
5 --> Kreuz
6 --> Raute
```

Größere Werte als 6 ergeben den Stern als Markierungsform. Werte für die Größe der Markierung werden in Pixel angegeben. Will man nur den zweiten oder dritten Parameter des Befehls benutzen, kann man z.B. den ersten Parameter fortlassen und nur die entsprechenden parametertrennenden Kommata setzen. DEFMARK „4 bedeutet z.B., daß die ersten beiden Parameter ihre aktuelle Einstellung behalten, aber die Größe der Markierungen auf 4 gesetzt wird.

Beispiel:

```
DIM x%(1),y%(1)
x%(0)=50
y%(0)=50
x%(1)=150
y%(1)=150
DEFMARK 1,4,2
POLYMARK 2,x%(),y%()
DEFMARK ,3,4
POLYMARK 2,x%(),y%() OFFSET 30,0
```

--> Zeichnet zwei Punktepaare mit unterschiedlichen Endpunktmarkierungen.

DEFFILL [farbe] , [stil] , [muster]
DEFFILL [farbe] , bitmuster$

```
farbe,stil,muster: iexp
bitmuster$: sexp
```

Dieser Befehl legt das Füllmuster für die Befehle PBOX, PCIRCLE, PELLIPSE, POLYFILL und FILL fest. Er bestimmt Farbe, Stil und Muster der Füllung und ermöglicht es, selbsterstellte Muster zu definieren. Für die Variable farbe können, je nach Auflösung, Werte von 0 bis 15 verwendet werden (siehe die Übersicht zu Beginn dieses Kapitels). Für stil gilt folgende Zuordnung:

```
0 --> Leer
1 --> Ausgefüllt
2 --> Punktiert
3 --> Schraffiert
4 --> ATARI-Symbol
```

Mit muster kann aus 24 Punkt- oder 12 Linienmustern eines ausgewählt werden (siehe Anhang: Füllmustertabelle).

In der zweiten Befehlsvariation wird in bitmuster$ die 32 Bytes umfassende Bit-Information zur Definition des 16*16 Pixel großen Füllmusters übergeben. Diese Informationen müssen im Wort-Format vorliegen, wozu sich der Befehl MKI$ anbietet.

Vorne stehende Parameter können weggelassen werden, wenn man die parametertrennenden Kommata aufführt. So wählt z.B. DEFFILL ,2,4 das Füllmuster 2,4 aus und läßt die Füllfarbe unverändert.

Bei Farbbetrieb können in der mittleren Auflösung den 16 Worten für die erste Bitplane noch 16 Worte für die zweite Bitplane folgen. In der niedrigen Auflösung sind es dann insgesamt 4 * 16 = 64 Worte für mehrfarbige Füllmuster.

Beispiele:

```
DEFFILL 1,2,4
PBOX 10,10,40,40
BOX 50,50,100,100
FILL 70,70
FOR i=1 TO 16
  f$=f$+MKI$(RAND(65535))
NEXT i
BOX 100,100,150,150
DEFFILL 1,f$
FILL 120,120
```

--> Zeichnet eine Box im voreingestellten und eine mit zufälligem Füllmuster.

```
DO
  FOR j%=0 TO 15
    f$=""
    s%=BCHG(s%,j%)
    FOR i%=1 TO 16
      f$=f$+MKI$(s%)
    NEXT i%
    DEFFILL 1,f$
    PBOX 0,0,639,399
  NEXT j%
LOOP
```

--> Definiert ein Füllmuster und gibt ein Rechteck mit diesem Muster aus.

```
FOR i%=1 TO 64
  READ a%
  a$=a$+MKI$(a%)
NEXT i%
DEFFILL ,a$
PBOX 20,20,300,200
DATA -1,-1,-1,-1,-1,-1,-1,-1, 0, 0, 0, 0, 0, 0, 0, 0
DATA -1,-1,-1,-1, 0, 0, 0, 0,-1,-1,-1,-1, 0, 0, 0, 0
DATA -1,-1, 0, 0,-1,-1, 0, 0,-1,-1, 0, 0,-1,-1, 0, 0
DATA -1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0
```

--> Diese Routine erstellt ein Füllmuster, das auf dem Schwarz-Weiß-Monitor zwei breite Streifen schwarz und weiß ausfüllt. Für den Farbmonitor ergeben sich bei mittlerer Auflösung vier halb so breite Streifen in den vier möglichen Farben und bei niedriger Auflösung 16 Streifen von jeweils einer Zeile Höhe in allen 16 Farben.

BOUNDARY n

n: iexp

Die Anweisung BOUNDARY benutzt die VDI-Funktion vsf_perimeter und schaltet die automatische Umrahmung der Füllfläche ein oder aus. Wenn n gleich Null ist, wird die Umrahmung ausgeschaltet und für n ungleich Null eingeschaltet.

Beispiel:

```
DEFFILL 1,2,2
BOUNDARY 1              ! Umrahmung einschalten
PBOX 50,50,100,100
BOUNDARY 0              ! Umrahmung ausschalten
PBOX 150,50,200,100
```

--> Zeichnet ein gefülltes Rechteck mit und ohne Umrahmung.

DEFLINE [stil] , [dicke] , [anfang_s,end_s]

```
stil,dicke,anfang_s,end_s: iexp
```

Der Befehl DEFLINE legt das Aussehen der Linien fest, die mit den Befehlen LINE, BOX, RBOX, CIRCLE, ELLIPSE und POLYLINE gezeichnet werden. Der erste Parameter stil legt den Linienstil fest. Hier besteht die Wahl zwischen vor- und selbstdefinierten Mustern. Folgende vordefinierte Muster sind möglich:

```
0 --> Linie in Hintergrundfarbe
1 --> Durchgezogene Linie
2 --> Gestrichelte Linie mit kleinen Abständen
3 --> Gepunktete Linie
4 --> Strichpunktierte Linie
5 --> Gestrichelte Linie mit großen Abständen
6 --> Das Linienmuster wird: Strich-Punkt-Punkt ...
```

Selbstdefinierte Muster können durch die Angabe eines 16 Bit-Wertes festgelegt werden, wobei ein gesetztes Bit einem gesetzten Punkt (im Monochrommodus) entspricht.

Der zweite Parameter dicke bestimmt die Dicke einer Linie in Pixel. Die Liniendicke kann nur ungerade Werte annehmen. Jede angegebene Liniendicke wird wie die nächst kleinere ungerade Zahl behandelt.

Mit anfang_s und end_s wird das Anfangs- und Endsymbol des Linientyps festgelegt. Hier besteht die Wahl zwischen:

```
0 --> Eckig
1 --> Pfeil
2 --> Rund
```

Vornestehende Parameter können fortgelassen werden, wenn man die parametertrennenden Kommata aufführt. So legt z.B. DEFLINE „1,1 als Symbol für Linienanfang und -ende den Pfeil fest, läßt aber Stil und Breite unverändert (siehe Tabelle im Anhang).

Beispiel:

```
FOR i=1 TO 6
  DEFLINE i
  LINE 50,i*50,200,i*50
NEXT i
DEFLINE 1,1,1,2
FOR i=2 TO 12 STEP 2
  DEFLINE ,i
  LINE 250,i*25,400,i*25
NEXT i
DEFLINE -&X101010101010101,1,0,0
LINE 500,10,500,390
VOID INP(2)
```

--> Es werden Linien in den sechs vordefinierten Stilarten gezeichnet. Dann werden Linien verschiedener Dicke erzeugt. Die Linie im selbstdefinerten Muster ist punktiert. Abschließend wird auf einen Tastendruck gewartet.

DEFTEXT [farbe],[attr],[winkel],[hoehe],[fontnr]

```
farbe,attr,winkel,hoehe,fontnr: iexp
```

Dieser Befehl legt das Aussehen einer mit TEXT auszugebenden Zeichenkette fest.

Dabei enthält farbe, je nach Auflösung, einen Wert zwischen 0 und 15. Mit attr werden folgende Textattribute festgelegt, die durch Addition der Kennziffern kombiniert werden können:

```
 0 --> normal        (normal)
 1 --> bold          (fett)
 2 --> light         (hell)
 4 --> italic        (schräg)
 8 --> underlined    (unterstrichen)
16 --> outlined      (umrandet)
```

winkel legt die Richtung der Textausgabe fest. Der Wert wird in 1/10 Grad gegen den Uhrzeigersinn drehend angegeben.

```
                 900
                  |
                  |
                  |
1800 ---- Textnullpunkt  ---- 0
                  |
                  |
                  |
                2700
```

Erlaubt sind die folgenden Werte:

```
   0 --> Voreinstellung, von links nach rechts
 900 --> Von unten nach oben
1800 --> Auf dem Kopf stehend, von rechts nach links
2700 --> Verdreht von oben nach unten
```

Weiter gibt hoehe die Texthöhe eines Großbuchstabens in Pixeln an. Dabei sind jedoch bei den normalen Zeichensätzen nur die folgenden Schrifthöhen gut lesbar:

```
 4 --> Icon-Schrifthöhe
 6 --> Kleine Schrift
13 --> Normale Schrifthöhe
32 --> Besonders große Schrifthöhe
```

Schließlich kann mit fontnr die Nummer eines gewünschten Zeichensatzes angegeben werden, der zuvor ordnungsgemäß installiert wurde (siehe auch GDOS: VST_LOAD_FONT, VQT_NAME, ...).

Beispiel:

```
FOR i|=0 TO 5
  DEFTEXT 1,2^i|,0,13
  TEXT 100,i|*16+100,"Dies ist das Textattribut "+STR$(i|)
NEXT i|
```

--> Gibt einen Beispieltext mit verschiedenen Attributen aus.

GRAPHMODE n

```
n: iexp
```

Der Befehl GRAPHMODE legt fest, in welcher Weise Grafikausgaben auf dem Bildschirm miteinander verknüpft werden. Dabei steht der numerische Ausdruck n für 4 mögliche Modi:

```
1 --> replace              (ersetzen)
2 --> transparent          (durchsichtig)
3 --> xor                  (invertieren)
4 --> reverse transparent  (invertiert und durchsichtig)
```

Hat n den Wert 1, dies ist der voreingestellte Modus, überdeckt das neue Muster das früher gezeichnete. Für n gleich 2 werden altes und neues Muster mit OR verknüpft.

Bei n gleich 3 wird jedes Pixel gesetzt, welches vorher nicht gesetzt war und jedes Pixel gelöscht, das vorher gesetzt war (die Muster werden also invertiert XOR-Verknüpfung).

Im Modus 4 wird das neue Muster zuerst invertiert und dann mit OR verknüpft.

Beispiel:

```
FOR i%=1 TO 4
  GRAPHMODE i%
  DEFFILL 1,3,8
  PBOX 150*i%-100,10,150*i%,100
  DEFFILL 1,2,10
  PBOX 150*i%-140,50,150*i%-40,150
NEXT i%
```

--> Das Beispiel zeigt 4 Rechteckpaare mit unterschiedlichen Mustern, die miteinander in den 4 Modi verknüpft sind.

Allgemeine Grafikbefehle

In diesem Abschnitt werden zunächst die Möglichkeiten zur Begrenzung von Grafikausgaben mit den CLIP-Befehlsvarianten aufgezeigt.

Die allgemeinen Grafikbefehle PLOT, LINE, BOX, RBOX, CIRCLE und ELLIPSE zeichnen Punkte, Linien, Rechtecke, Rechtecke mit abgerundeten Ecken, Kreise und Ellipsen. Mit Hilfe von PBOX, PRBOX, PCIRCLE und PELLIPSE können sie mit Farben oder Mustern gefüllt werden.

POLYLINE zeichnet einen Polygonzug, dessen Eckpunkten mit POLYMARK verschiedene Symbole zugeordnet werden können. POLYFILL füllt dieses Polygon mit einem Muster in einer bestimmten Farbe. POINT liefert die Farbe eines bestimmten Bildschirmpunktes. FILL füllt einen umrandeten Bildschirmbereich. TEXT ermöglicht die Ausgabe von Zeichenketten an beliebigen Stellen auf dem Bildschirm. CLS löscht den Bildschirm. Der Koordinatenursprung liegt bei diesen Grafikanweisungen in der linken oberen Bildschirmecke. Die Koordinaten können Werte außerhalb der Bildschirmauflösung annehmen, jedoch werden nur die sichtbaren Teile der Grafik dargestellt. Am Ende des Abschnitts wird der Befehl BITBLT vorgestellt.

CLIP x,y,w,h [**OFFSET** x0,y0]
CLIP x1,y1 **TO** x2,y2 [**OFFSET** x0,y0]
CLIP #n [**OFFSET** x0,y0]
CLIP OFFSET x,y
CLIP OFF

```
x,y,w,h,x0,y0,x1,y1,x2,y2,n: iexp
```

Diese Gruppe von Anweisungen dient dem 'Clipping'. Darunter versteht man die Begrenzung von Grafikausgaben auf einen beliebigen, rechteckigen Bildschirmausschnitt. CLIP setzt das 'Clipping' für die VDI-Grafikbefehle, während ACLIP dies nur für Line-A-Grafikroutinen tut. Zur Festlegung dieses Bildschirmausschnittes (clipping rectangle) müssen die Koordinaten der diagonal gegenüberliegenden Eckpunkte bzw. Breite und Höhe des Begrenzungsrechtecks übergeben werden.

Dabei werden verschiedene Varianten des CLIP-Befehls unterschieden. CLIP x,y,w,h ermöglicht die Angabe der linken x-Koordinate 'x' und der oberen y-Koordinate 'y' sowie Breite 'w' und Höhe 'h' des Begrenzungsrechtecks.

Eine weitere Möglichkeit bietet der Befehl CLIP x1,y1 TO x2,y2. Hierbei werden die Koordianten der diagonal gegenüberliegenden Eckpunkte (x1,y1) und (x2,y2) übergeben.

Die dritte Variante ermöglicht die Begrenzung der Ausgaben auf die Ausdehnung des Fensters mit der Nummer 'n'.

Bei den genannten Varianten ist es mit dem optionalen Befehlszusatz OFFSET x0,y0 möglich, den Ursprung für die Grafikausgaben auf den Punkt mit den Koordianten x0,y0 festzulegen. Weiterhin kann der Befehl OFFSET x0,y0 auch alleine stehen, wobei der gleiche Zweck erfüllt wird und der Ursprung für die Grafikausgaben ebenso auf den Punkt (x0,y0) gelegt wird. Das 'clipping' wird mit Hilfe des Befehls CLIP OFF abgeschaltet. Die Begrenzung der Grafikausgaben gilt nicht für die Befehle GET, PUT, BITBLT sowie alle Line-A-Aufrufe und AES-Befehle.

PLOT x,y
LINE x1,y1,x2,y2
DRAW [**TO**] [x,y]
DRAW [x1,y1] [**TO** x2,y2] [**TO** x3,y3] [**TO** ...]

```
x,y,x1,y1,x2,y2: iexp
```

PLOT zeichnet einen Punkt mit den Koordinaten x,y auf den Bildschirm. LINE zeichnet eine Linie zwischen den Koordinatenpaaren x1,y1 und x2,y2. Stil und Farbe dieser Linie können mit DEFLINE und COLOR festgelegt werden.

DRAW x,y entspricht dem Befehl PLOT. Mit Hilfe von DRAW TO x,y wird eine Linie zwischen den Koordinaten x,y und dem letzten gesetzten Punkt gezeichnet. Dabei ist es gleichgültig, ob dieser Punkt durch PLOT, LINE oder DRAW gesetzt wurde. Eine weitere Variante des DRAW-Befehls stellt DRAW x1,y1 TO x2,y2 dar. Dies entspricht dem LINE-Befehl. Zusätzlich können weitere Koordinatenangaben angehängt werden. Damit können z.B. Vielecke erzeugt werden. Schließlich ermöglicht die letzte Variante des DRAW-Befehls die Angabe von Kommandos, die Ähnlichkeit mit bestimmten Grafik-Kommandos von LOGO (Turtle-Grafik) und der Standard-Plottersprache HPGL von Hewlett-Packard haben. Damit ist es möglich, einen Plotter auf dem Bildschirm zu simulieren.

Beispiele:

```
x=50
y=50
farbe=POINT(x,2*y)
PRINT farbe
PLOT x,2*50
LINE 200,200,400,100
PRINT POINT(x,100)
```

--> Zeichnet einen Punkt und ermittelt dessen Zeichenfarbe.

```
DO
  MOUSE mx,my,mk
  IF mk=1
    DRAW TO mx,my
  ENDIF
  EXIT IF mk=2
LOOP
```

--> Bei gedrückter linker Maustaste wird eine Verbindung zwischen den aktuellen Mauskoordinaten mx,my und dem letzten gesetzten Punkt gezeichnet. Die Schleife wird durch Drücken der rechten Maustaste verlassen.

DRAW ausdruck
DRAW(i)
SETDRAW

```
i: iexp
ausdruck: eine Folge von sexp und aexp, der erste muß sexp sein,
          Trennung durch Komma, Semikolon oder Hochkomma
```

Mit DRAW wird ein imaginärer Stift über den Bildschirm bewegt und kann relativ zeichnen. Dies ähnelt sehr den Grafikbefehlen der Programmiersprache Logo. Die Parameter der einzelnen Befehle sind jeweils Fließkommazahlen, die auch in Strings angegeben werden können. Es gibt die Kommandos:

FD n	ForwarD	Bewegt den 'Stift' n Pixel 'vorwärts'.
BK n	BacKward	Bewegt den 'Stift' n Pixel 'rückwärts'.
SX x	Scale X	Skaliert die 'Stiftbewegung' für FD und BK
SY y	Scale Y	mit dem angegebenen Faktor. Die Skalierung mit SX und SY wirkt nur auf die Befehle FD und BK. Mit SX0 bzw. SY0 wird die Skalierung ausgeschaltet (schneller als die Skalierung mit dem Faktor 1 (SX1, SY1)).
LT w	Left Turn	Gibt den Winkel 'w' an, um den die Zeichenrichtung nach links gedreht wird.
RT w	Rigth Turn	Analog nach rechts.
TT w	Turn To	Gibt den absoluten Winkel 'w' an, dazu gilt folgende Zuordnung:

```
                0
                |
                |
     270 ---- Nullpunkt ---- 90
                |
                |
               180
```

Die Angaben für w = winkel erfolgen in Grad.

MA x,y	Move Absolute	Bewegt den 'Stift' an die absoluten Koordinaten x und y.
DA x,y	Draw Absolute	Bewegt den 'Stift' an die absoluten Koordinaten x und y und zeichnet eine Linie in der aktuellen Farbe von der letzten Position zum Punkt (x,y).
MR xr,yr	Move Relative	Wie 'MA', jedoch in Bezug zur letzten Position.
DR xr,yr	Draw Relative	Wie 'DR', jedoch in Bezug zur letzten Position.

Eine Verkürzung für den Ausdruck DRAW "MA",x,y,"TT",w stellt der Befehl SETDRAW x,y,w dar.

CO c	Color	Setzt die Farbe 'c' als 'Zeichenfarbe' (vgl. auch Parameter bei COLOR-Befehl).
PU	Pen Up	Hebt den 'Stift'.
PD	Pen Down	Senkt den 'Stift'.

Zusätzlich stehen die folgenden Nachfragefunktionen zur Verfügung:

DRAW(0)	liefert	x-Position (als Fließkommawert)
DRAW(1)	liefert	y-Position (als Fließkommawert)
DRAW(2)	liefert	Winkel in Grad (als Fließkommawert)
DRAW(3)	liefert	Skalierung der x-Achse (als Fließkommawert)
DRAW(4)	liefert	Skalierung der y-Achse (als Fließkommawert)
DRAW(5)	liefert	Penflag (-1=PD, 0=PU)

Beispiele:

```
DRAW "ma 160,200 tt0"     ! starte bei 160,200 mit dem Winkel 0
FOR i&=3 TO 10
  eck(i&,90)              ! zeichne ein Polygon mit i-Ecken
NEXT i&
'
PROCEDURE eck(n&,r&)      ! n=anzahl der Ecken, r=Kantenlänge
  LOCAL i&
  FOR i&=1 TO n&
    DRAW "fd",r&,"rt",360/n&
  NEXT i&
RETURN
```

--> Zeichnet Vielecke.

```
FOR i=0 TO 359 STEP 8
  SETDRAW 320,200,i
  GRAPHMODE 3
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
  GRAPHMODE 1
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
NEXT i
```

--> Zeichnet ein kleines und ein großes Rechteck, die sich um ihre eigene Achse drehen.

```
l%=48
' raute 1
DRAW "ma60,100 tt45"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' rhomboid, schmal
DRAW "mr100,0 tt45"
DRAW "sx0.5 sy0"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' raute, breit
DRAW "mr100,0 tt45"
DRAW "sx0 sy0.5"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' raute, breit und hoch
DRAW "mr100,0 tt45"
DRAW "sx3 sy2"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
```

--> Zeichnet drei Rauten und einen Rhomboiden, dazu wird als Ausgangsobjekt eine Raute benutzt und die anderen Figuren durch Veränderung der x- bzw. y-Skalierung erreicht.

BOX x1,y1,x2,y2
PBOX x1,y1,x2,y2
RBOX x1,y1,x2,y2
PRBOX x1,y1,x2,y2

```
x1,y1,x2,y2: iexp
```

BOX zeichnet ein Rechteck mit den diagonal gegenüberliegenden Eckpunktkoordinaten x1,y1 und x2,y2. Analog dazu zeichnen PBOX ein gefülltes Rechteck, RBOX ein Rechteck mit abgerundeten Ecken und PRBOX ein gefülltes Rechteck mit abgerundeten Ecken.

Beispiel:

```
BOX 20,20,120,120
RBOX 170,20,270,120
x=150
DEFFILL 1,2,4
PBOX 20,20+x,120,120+x
PRBOX 170,20+x,270,120+x
~INP(2)
```

--> Zeichnet ein Rechteck, dann ein Rechteck mit abgerundeten Ecken und schließlich beide Formen noch einmal ausgefüllt. Abschließend wird auf einen Tastendruck gewartet.

CIRCLE x,y,r [,w1,w2]
PCIRCLE x,y,r[,w1,w2]
ELLIPSE x,y,rx,ry[,w1,w2]
PELLIPSE x,y,rx,ry[,w1,w2]

```
x,y,r,w1,w2: iexp
```

CIRCLE zeichnet einen Kreis um den Mittelpunkt mit den Koordinaten x,y und dem Radius r. Zusätzlich können Anfangswinkel w1 und Endwinkel w2 zur Bestimmung eines Kreisbogens angegeben werden. Analog dazu zeichnet PCIRCLE einen gefüllten Kreis oder ein gefülltes Kreissegment.

ELLIPSE zeichnet eine Ellipse mit den Mittelpunkt-Koordinaten x,y und dem waagerechten Radius rx sowie dem senkrechten Radius ry. Zusätzlich können Anfangswinkel w1 und Endwinkel w2 zur Bestimmung eines Ellipsenbogens angegeben werden. Analog dazu zeichnet PELLIPSE eine gefüllte Ellipse oder ein gefülltes Ellipsensegment. Dabei werden die Winkel in 1/10 Grad angegeben (0-3600) und die Kreisbögen oder -segmente gegen den Uhrzeigersinn verlaufend gezeichnet (d.h. 0 Grad rechts, 90 Grad oben, 180 Grad links und 270 Grad unten).

Beispiel:

```
CIRCLE 320,200,100
ELLIPSE 320,200,200,100,900,1800
PCIRCLE 320,200,100,1800,2700
PELLIPSE 320,200,200,100,2700,3600
```

--> Zeichnet einige Kreis- und Ellipsen(-Teile).

POLYLINE n,x(),y() [**OFFSET** x_off,y_off]
POLYMARK n,x(),y() [**OFFSET** x_off,y_off]
POLYFILL n,x(),y() [**OFFSET** x_off,y_off]

```
n,x_off,y_off: iexp
x(),y(): avar-Array
```

POLYLINE zeichnet einen Linienzug mit n Ecken. Die x,y-Koordinaten der Eckpunkte stehen in den Feldern x() und y(). Der erste Eckpunkt steht in x(0) und y(0), der letzte in x(n-1) und y(n-1). Der erste und der letzte Eckpunkt werden automatisch miteinander verbunden. Zusätzlich kann ein waagerechter (x_off) oder senkrechter (y_off) OFFSET auf diese Koordinaten addiert werden.

POLYFILL füllt das Vieleck mit einem Muster und einer Farbe, die mit DEFFILL festgelegt werden können.

Mit POLYMARK können die Punkte markiert werden.

Beispiel:

```
DIM x%(3),y%(3)
FOR i%=0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
DATA 120,120,170,170,70,170,120,120
POLYLINE 4,x%(),y%()
POLYFILL 3,x%(),y%() OFFSET -50,-50
DEFMARK ,4,10
POLYMARK 3,x%(),y%() OFFSET 40,-80
~INP(2)
```

--> Zeichnet ein nicht ausgefülltes und ein ausgefülltes Dreieck sowie rechteckige Eckpunktmarkierungen eines weiteren Dreiecks.

POINT(x,y)

```
x,y: iexp
```

Die Farbe des Punktes mit den Koordinaten x,y wird ermittelt und der Variablen zurückgemeldet. Hierbei werden in der niedrigen Auflösung Werte zwischen 0 und 15, in der mittleren von 0 bis 3 und in der hohen Auflösung nur 0 oder 1 zurückgeliefert.

Beispiel:

```
a=POINT(100,100)
PLOT 100,100
PRINT a,POINT(100,100)
```

--> Schreibt die Farbe an der Bildschirmkoordinate 100,100 vor und nach dem PLOT-Befehl auf den Bildschirm.

FILL x,y [,f]

```
x,y,f: iexp
```

Dieser Befehl füllt eine beliebig geformte Fläche. Der Füllvorgang beginnt an den Koordinaten x,y. Bei Angabe von f wird der Füllvorgang durch Punkte der Farbe f und dem Bildschirmrand begrenzt.

Ohne Angabe von f oder mit f=-1 wird jeder Punkt mit anderer Farbe als der Startpunkt (x,y) als Begrenzung benutzt.

Beispiele:

```
LINE 0,180,639,180
FOR i=1 TO 19
  BOX i*20,100,i*20+20-i,180
  TEXT i*20-4,195,i
  DEFFILL ,2,i
  FILL i*20+1,101
NEXT i
```

--> Zeichnet eine Gerade, auf der sich Rechtecke mit verschiedenen Mustern aneinanderreihen und füllt diese ohne Farbbegrenzung, was zur Zerstörung der Muster führt.

```
LINE 0,280,639,280
FOR i=1 TO 19
  BOX i*20,200,i*20+20-i,280
  TEXT i*20-4,295,i
  DEFFILL ,2,i
  FILL i*20+1,201,1
NEXT i
```

--> Zeichnet darunter ebenfalls eine Gerade mit aufeinanderfolgenden unterschiedlich gemusterten Rechtecken und füllt mit Farbbegrenzung. Dadurch bleiben die verschiedenen Muster erhalten.

CLS[#n]

n: iexp

Löscht den Bildschirm durch Ausgabe von ESC-E-CR. Kann auch auf Dateien umgeleitet werden.

Beispiel:

```
PBOX 100,100,500,200
REPEAT
UNTIL MOUSEK
CLS
```

--> Füllt den Bildschirm teilweise mit einem Rechteck und löscht ihn nach dem Drücken einer Maustaste.

TEXT x,y [,l] , ausdruck

```
x,y,l: iexp
ausdruck: sexp oder aexp
```

Gibt am Punkt mit den Koordinaten x,y den ausdruck als Grafiktext aus. Dieser Punkt bezieht sich auf die linke Ecke der Grundlinie des ersten Zeichens von ausdruck. Der Parameter l legt die Länge der Textausgabe in Pixel fest. Dies geschieht durch Veränderung des Zeichenabstandes (bei l>0) oder durch Veränderung des Wortabstandes (bei l<0). Ist l=0, erfolgt die unveränderte Ausgabe von ausdruck.

Der Grafiktext kann durch DEFTEXT mit Attributen versehen werden. Jedoch wirkt DEFTEXT nur auf den TEXT-Befehl und auf den PRINT-Befehl innerhalb eines Fensters.

Beispiel:

```
s$="Dies ist ein Beispiel."
FOR i=0 TO 23
  DEFTEXT 1,i,0,6
  TEXT 50,i*16+16,s$
NEXT i
DEFTEXT 1,0,0,13
TEXT 350,50,350-50
TEXT 350,100,s$
TEXT 350,150,250,s$
TEXT 350,200,-250,s$
~INP(2)
```

--> Schreibt Grafiktexte in verschiedenen Varianten auf den Bildschirm und wartet dann auf einen Tastendruck.

SPRITE bitmuster$ [,x,y]

```
bitmuster$: svar
```

Der Befehl SPRITE ermöglicht es, ein 16 mal 16 Pixel großes Objekt über den Bildschirm zu bewegen. Die entsprechenden Bit-Informationen für das Muster und die Maske werden in einem String abgelegt. Dabei muß beachtet werden, daß alle diese Werte in Wortgröße übergeben werden müssen. Zu diesem Zweck kann man den Befehl MKI$ verwenden, so daß sich bitmuster$ wie folgt ergibt:

```
bitmuster$=MKI$(x-Koordinate Aktionspunkt)
          +MKI$(y-Koordinate Aktionspunkt)
          +MKI$(0) für normal oder MKI$(-1) für XOR
          +MKI$(Maskenfarbe) meist 0
          +MKI$(Spritefarbe) meist 1
          +sprite$
```

Dabei enthält sprite$ die Bit-Informationen für Spriteform und -maske, die im Gegensatz zu DEFMOUSE nicht nacheinander, sondern wechselweise angegeben werden müssen.

Beispiel:

```
gfa$=MKI$(1)+MKI$(1)+MKI$(0)
gfa$=gfa$+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  READ muster%,maske%
  gfa$=gfa$+MKI$(maske%)+MKI$(muster%)
NEXT i%
DATA 0,0,0,32256,15360,16896,8192,24064
DATA 8192,24560,11744,21008,9472,23280,9472,23295
DATA 15838,16929,274,32493,274,749,286,737
DATA 18,1005,18,45,18,45,0,63
```

```
REPEAT
  ADD mx%,(MOUSEX-mx%)/50
  ADD my%,(MOUSEY-my%)/50
  SPRITE gfa$,mx%,my%
UNTIL MOUSEK=2
```

--> Bewegt ein Sprite über den Bildschirm, das träge dem Maus-Cursor folgt.

Bildschirmausschnitte

SGET bildschirm$
SPUT bildschirm$

```
bildschirm$: svar
```

SGET kopiert den gesamten Bildschirm (32000 Byte) besonders schnell in einen String. Analog dazu kopiert SPUT einen 32000 Bytes langen String in den Bildschirmspeicher.

```
PCIRCLE 100,100,50
SGET b$
~INP(2)
CLS
~INP(2)
SPUT b$
```

--> Malt einen ausgefüllten Kreis auf den Bildschirm. Nach einem Tastendruck verschwindet er, nach einem weiteren Tastendruck ist er wieder sichtbar.

GET x1,y1,x2,y2,ausschnitt$
PUT x1,y1,ausschnitt$ [,modus]

```
x1,y1,x2,y2,modus: iexp
ausschnitt$: svar
```

GET legt einen Bildschirmausschnitt in einer String-Variablen ab. Analog legt PUT einen mit GET abgelegten Bildschirmausschnitt an der Stelle x,y in den Bildschirmspeicher. Mit modus kann optional bestimmt werden, wie das in ausschnitt$ abgelegte Bit-Muster mit dem bestehenden Bildschirminhalt verknüpft wird. In nachfolgender Tabelle bezeichnet Q (Quelle) das im String enthaltene Bit-Muster und Z (Ziel) den ursprünglichen Bildschirminhalt.

Modus	Verknüpfungsregel	Wirkung
0	0	Alle Punkte werden gelöscht.
1	Q AND Z	Es bleiben nur die Punkte gesetzt, die in beiden Rastern gesetzt sind.
2	Q AND (NOT Z)	Es werden nur die Punkte gesetzt, die im Quellraster gesetzt und im Zielraster gelöscht sind.
3	Q	Das Quellraster wird unverändert übertragen (GRAPHMODE 1, Voreinstellung).
4	(NOT Q) AND Z	Es werden nur die Punkte gesetzt, die im Quellraster gelöscht und im Zielraster gesetzt sind.
5	Z	Das Ziel bleibt unverändert.
6	Q XOR Z	Es werden nur die Punkte gesetzt, die in genau einem Raster sind (GRAPHMODE 3).
7	Q OR Z	Es werden alle Punkte gesetzt, die in einem der beiden Raster gesetzt sind (GRAPHMODE 2).
8	NOT (Q OR Z)	Es werden alle Punkte gesetzt, die in keinem der beiden Raster gesetzt sind.

9	NOT (Q XOR Z)	Es werden alle Punkte gesetzt, die in beiden Rastern oder in keinem Raster gesetzt sind.
10	NOT Z	Das Zielraster wird invertiert.
11	Q OR (NOT B)	Es werden alle Punkte gesetzt, die im Quellraster gesetzt sind und alle, die im Zielraster nicht gesetzt waren.
12	NOT B	Das Quellraster wird vor der Übertragung invertiert.
13	(NOT Q) OR B	GRAPHMODE 4
14	NOT (Q AND Z)	Es werden alle Punkte gesetzt, die nicht in beiden Rastern enthalten waren.
15	1	Es werden alle Punkte gesetzt.

Wird außerdem Bit 4 des Modus gesetzt, so erfolgt zusätzlich eine AND-Verknüpfung des Füllmusters mit dem Quellraster.

VSYNC

Dieser Befehl dient der Synchronisation des Bildschirmaufbaus (d.h. das Programm wartet auf senkrechten Strahlrücklauf des Monitors, 'vertical blank interrupt').

VSYNC kann z.B. bei der Animation von Bildschirmteilen mit GET und PUT verwendet werden.

Beispiel:

```
t%=TIMER
FOR i%=1 TO 100
  VSYNC
NEXT i%
PRINT SUB(TIMER,t%)/200
```

--> Gibt die Zeit für 100 Bildwechsel aus.

BITBLT s_mfdb%(),d_mfdb%(),par%()

```
s_mfdb%(),d_mfdb%(),par%(): Integer-Arrays
```

Der Befehl BITBLT dient dem Kopieren von rechteckigen Bildschirmausschnitten. Er hat Ähnlichkeit mit den Befehlen GET und PUT, ist jedoch flexibler und schneller, wenn auch komplizierter in der Anwendung.

Die Parameter des Befehls werden in drei Feldern abgelegt. Das erste beschreibt den Aufbau des zu kopierenden Bereichs (Quellraster) und das zweite den Bereich, in dem die Kopie abgelegt werden soll (Zielraster). Das dritte enthält die Koordinaten des Quell- und Zielbereichs sowie den Kopiermodus.

Diesem Befehl liegt eine VDI-Routine zu Grunde, während BITBLT adr% und BITBLT x%() eine LINE-A-Routine benutzen (siehe Abschnitt LINE-A-Aufrufe).

Der Aufbau des Quell- (s_mfdb%) und des Zielrasters (d_mfdb%) ist gleich. Die Abkürzungen bedeuten:

```
s_mfdb%() source memory form description block
d_mfdb%() destination memory form description block
```

Die Array-Elemente sind:

```
_mfdb%(0)  Anfangsadresse des Rasters. Diese Adresse muß gerade sein.
_mfdb%(1)  Breite des Rasters in Pixeln. Dieser Wert muß durch 16
           teilbar sein.
_mfdb%(2)  Höhe des Rasters in Pixeln.
_mfdb%(3)  Rasterbreite in Worten (= Pixelanzahl / 16).
_mfdb%(4)  Reserviert, immer 0.
_mfdb%(5)  Anzahl der Rasterebenen:
            Hohe Auflösung = 1
            Mittlere Auflösung = 2
            Niedrige Auflösung = 4
```

Das Feld par%() hat folgenden Aufbau:

```
par%(0)  Linke x-Koordinate des Quellrasters.
par%(1)  Obere y-Koordinate des Quellrasters.
par%(2)  Rechte x-Koordinate des Quellrasters.
par%(3)  Untere y-Koordinate des Quellrasters.
par%(4)  Linke x-Koordinate des Zielrasters.
par%(5)  Obere y-Koordinate des Zielrasters.
par%(6)  Rechte x-Koordinate des Zielrasters.
par%(7)  Untere y-Koordinate des Zielrasters.
par%(8)  Kopiermodus.
```

Die Werte für den Kopiermodus entsprechen denen bei GET/PUT (siehe auch im Kapitel Grafik). Die wichtigsten sind:

```
  3 = replace              (GRAPHMODE 1)
  6 = xor                  (GRAPHMODE 2)
  7 = transparent          (GRAPHMODE 3)
 13 = invers transparent   (GRAPHMODE 4)
```

Beispiel:

```
DIM smfdb%(8),dmfdb%(8),p%(8)
'
FOR i%=0 TO 639 STEP 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6       !nur wenn auch etwas da ist
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
```

```
    smfdb%(0)=a%+6
    smfdb%(1)=(b%+16) AND &HFFF0
    smfdb%(2)=h%+1
    smfdb%(3)=smfdb%(1)/16
    smfdb%(5)=DPEEK(a%+4)
    dmfdb%(0)=XBIOS(3)
    '
    dmfdb%(1)=640
    dmfdb%(2)=400
    dmfdb%(3)=40
    dmfdb%(5)=1
    '
    p%(1)=0
    p%(3)=h%
    p%(5)=y%
    p%(7)=y%+h%
    p%(8)=3
    p%(4)=x%+b%
    p%(6)=x%+b%
    '
    FOR i%=0 TO b%
      p%(0)=i%
      p%(2)=i%
      BITBLT smfdb%(),dmfdb%(),p%()
      DEC p%(4)
      DEC p%(6)
    NEXT i%
    '
  ENDIF
RETURN
```

--> Zeichnet einen großen Kreis, in dem ein kleiner Kreis liegt. Anschließend wird der gesamte Bildschirm in einen String eingelesen und daraufhin an der senkrechten Achse gespiegelt (vgl. BITBLT im Abschnitt LINE-A-Aufrufe).

9 - Ereignis-, Menü- und Fensterverwaltung

Ereignisverwaltung

Es gibt spezifische GFA-BASIC-Befehle, mit deren Hilfe man auf einfache Art Ereignisse (in der Literatur zu GEM Events genannt) abfragen kann. Diese Ereignisse sind Maustastendruck, Tastendruck, die Überwachung der Mauskoordinaten in Relation zu zwei rechteckigen Bildschirmbereichen und das Eintreffen einer "GEM-message", in der sich z.B. Informationen zur Fensterverwaltung befinden.

Die Überwachung dieser Ereignisse wird durch ein ON MENU xxx GOSUB eingeschaltet, wobei xxx das Ereignis ist, auf das reagiert werden soll. Die Überwachung findet mit Hilfe des ON MENU-Befehls statt. Bei jedem Aufruf dieses Befehls wird geprüft, ob ein Ereignis eingetreten ist. Ist ein Ereignis aufgetaucht, auf das reagiert werden soll, so wird zur dafür festgelegten Prozedur verzweigt.

ON MENU [t]

t: iexp

Der Befehl ON MENU überwacht das Auftreten von Ereignissen. Vor diesem Befehl sollte festgelegt werden, in welche Prozedur das Programm bei welchem Ereignis verzweigen sollte. Die Befehle, mit denen dies festgelegt wird, werden im Rest dieses Kapitels beschrieben. Für ein ständiges Überwachen eines Ereignisses ist es notwendig, diesen Befehl wiederholt zu durchlaufen. Daher findet sich der ON MENU-Befehl normalerweise in einer Schleife.

Der Parameter t enthält die Zeit (in tausendstel Sekunden), nach der der ON MENU-Befehl beendet sein soll. Der Sinn dieser Zeitangabe besteht darin, daß das GEM gelegentlich das Loslassen eines Mausknopfes nicht bemerkt (typischer Effekt: Schließfeld eines Fensters invertiert, Fenster bleibt aber offen, bis man mehrfach energisch nachklickt), was bei der Benutzung dieser Befehlsvariante nicht mehr passieren sollte. Die Voraussetzung dafür ist jedoch, daß die Maustasten auch überwacht werden. Dazu benötigt man einen ON MENU BUTTON x,y,z GOSUB-Befehl.

Beispiel:

```
ON MENU BUTTON 1,1,1 GOSUB test
t%=TIMER
REPEAT
  PRINT (TIMER-t%)/200
  ON MENU 2000
UNTIL MOUSEK=2
PROCEDURE test
RETURN
```

--> Gibt alle zwei Sekunden die Zeit seit Programmstart aus. Wird die linke Maustaste gedrückt, so ist das zu entdeckende Ereignis eingetreten, und ON MENU wird vor Ablauf der Zwei-Sekunden-Frist beendet. Der Druck der rechten Maustaste führt zum Programmabbruch. Ändert man die erste Zeile in: ON MENU BUTTON 0,0,0 GOSUB test, so wird die Mausüberwachung abgeschaltet, und der Zeitparameter hinter ON MENU ist effektlos.

MENU(x)

```
x: aexp zwischen -2 und 15 einschließlich
```

In den Variablen MENU(-2) bis MENU(15) finden sich sich alle relevanten Informationen für den Umgang mit Ereignissen. Für den Fall, daß ein Menüeintrag in einem Pull-Down-Menü gewählt wurde, steht in MENU(0) der Index des gewählten Eintrags im Array der Menüeinträge (siehe nächsten Abschnitt und Beispielprogramm am Ende des Abschnitts über Pull-Down-Menüs).

MENU(-2) enthält die Adresse des Message-Buffers, und in MENU(-1) findet sich die Adresse des Objektbaums des Pull-Down-Menüs.

In den Variablen MENU(1) bis MENU(8) liegt der Message-Buffer und in MENU(9) bis MENU(15) der AES-Integer-Output-Block. Die Nutzung der an diesen Stellen zu findenden Informationen soll nun kurz diskutiert werden.

Ausgangspunkt dieser Erläuterung ist MENU(1) und der Message-Buffer. In MENU(1) steht die Kennummer des aufgetretenen Ereignisses. In Abhängigkeit von MENU(1) enthalten die anderen Elemente des Message-Buffers verschiedene Informationen. Die folgende Tabelle faßt diese Informationen zusammen. Es ist jeweils der Wert von MENU(1) aufgeführt, dahinter die Bedeutung dieses Wertes und schließlich die Variablen, in denen sich bei diesem MENU(1)-Wert wichtige Informationen befinden. Diese Informationen sind vor allem für die korrekte Verwaltung von Fenstern wichtig:

```
MENU(1) = 10   Ein Pull-Down-Menüeintrag wurde angewählt.
      MENU(0)  Menüeintrag-Index im Eintragarray.
      MENU(4)  Objekt-Index des Menütitels.
      MENU(5)  Objekt-Index des gewählten Menü-Eintrags.
```

MENU(1) = 20 Ein rechteckiger Fensterausschnitt muß neu gezeichnet werden.
MENU(4) Kennung (Handle) des Fensters.
MENU(5) linke x-Koordinate des Ausschnitts.
MENU(6) obere y-Koordinate des Auschnitts.
MENU(7) Breite des Ausschnitts.
MENU(8) Höhe des Ausschnitts
(Beispiel bei ON MENU MESSAGE GOSUB).

MENU(1) = 21 Ein Fenster wurde angeklickt (normalerweise bedeutet dies, daß der Benutzer dieses Fenster aktivieren will).
MENU(4) Kennung (Handle) des angeklickten Fensters.

MENU(1) = 22 Das Schließfeld eines Fensters wurde angeklickt.
MENU(4) Kennung (Handle) dieses Fensters.

MENU(1) = 23 Das rechte obere Feld eines Fensters wurde angeklickt (normalerweise bedeutet dies, daß der Benutzer das Fenster auf Maximalgröße bringen will).
MENU(4) Kennung (Handle) dieses Fensters.

MENU(1) = 24 Es wurde eines der vier Pfeilfelder oder ein Schiebereglerbereich des Fensterrahmens angeklickt. Das Verschieben eines Reglers führt zu MENU(1)=25, bzw. 26. Hier wird nur geprüft, ob rechts oder links von der aktuellen Schiebereglerposition in den gemusterten Bereich geklickt wurde.
MENU(4) Kennung (Handle) des Fensters.
MENU(5) Was wurde angeklickt ? Es gilt die Zuordnung:
0: Oberhalb des rechten Reglers.
1: Unterhalb des rechten Reglers.
2: Pfeil nach oben.
3: Pfeil nach unten.
4: Links vom unteren Regler.
5: Rechts vom unteren Regler.
6: Pfeil nach links.
7: Pfeil nach rechts.

MENU(1) = 25 Der untere Schieberegler wurde verschoben.
MENU(4) Kennung (Handle) des Fensters.
MENU(5) Position des Reglers (Zahl zwischen 1 und 1000).

MENU(1) = 26 Der rechte Schieberegler wurde verschoben.
MENU(4) Kennung (Handle) des Fensters.
MENU(5) Position des Reglers (Zahl zwischen 1 und 1000).

MENU(1) = 27 Die Größe des Fensters wurde mit Hilfe des rechten, unteren Bedienelements verändert. Die neue Größe des Fensters wird zurückgemeldet in:
MENU(4) Kennung (Handle) des Fensters.
MENU(5) Linke X-Koordinate.
MENU(6) Obere Y-Koordinate.
MENU(7) Breite des Fensters.
MENU(8) Höhe des Fensters.

MENU(1) = 28 Die Position eines Fensters wurde verändert, die neuen Koordinaten stehen in:
MENU(4) Kennung (Handle) des Fensters.
MENU(5) Linke X-Koordinate.
MENU(6) Obere Y-Koordinate.
MENU(7) Breite des Fensters.
MENU(8) Höhe des Fensters.

MENU(1) = 29 Ein neues Fenster wurde vom GEM aktiviert. Dies kann beim Schließen eines darüberliegenden Fensters geschehen, in dem z.B. ein Accessory lief.
MENU(4) Kennung (Handle) des Fensters.

MENU(1) = 40 Ein Accessory wurde gewählt. Dieser Wert kann nur von einem Accessory empfangen werden, das an Hand des in MENU(4) stehenden Wertes prüfen muß, ob es aufgerufen wurde.
MENU(4) Menü-Identifikationsnummer des Accessories.

MENU(1) = 41 Ein Accessory wurde geschlossen. Dieser Wert kann nur von einem Accessory empfangen werden, das an Hand des in MENU(4) stehenden Wertes prüfen muß, ob es beendet wurde.

MENU(4) Menü-Identifikationsnummer des Accessories.

Die Variablen MENU(9) bis MENU(15) und GINTOUT(0) bis GINTOUT(7) enthalten, wenn in MENU(9) das Bit für das entsprechende Ereignis gesetzt ist, folgende Informationen:

In MENU(9) steht in jeweils einem Kenn-Bit, welche Art von Ereignis eingetreten ist. Die Zuordnung ist:

```
Bit 0 --> Tastatur
Bit 1 --> Maustaste
Bit 2 --> Maus hat Rechteck 1 verlassen/betreten.
Bit 3 --> Maus hat Rechteck 2 verlassen/betreten.
Bit 4 --> Eine Nachricht ist im Message-Buffer eingetroffen.
Bit 5 --> Zeitgeber (timer).
```

MENU(10) X-Position der Maus bei Ereignisauslösung.

MENU(11) Y-Position der Maus bei Ereignisauslösung.

MENU(12) Maustastenstatus, es gilt:

```
 0 --> Keine Taste gedrückt.
 1 --> Linke Taste gedrückt.
 2 --> Rechte Taste gedrückt.
 3 --> Beide Tasten gedrückt.
       (Beispiel bei ON MENU BUTTON x,y,z GOSUB)
```

MENU(13) Status der Tastaturumschalttasten; für jede gedrückte Umschalttaste wird ein Bit gesetzt. Es gilt die Zuordnung:

```
 Bit 0 --> Rechte Shift-Taste
 Bit 1 --> Linke Shift-Taste
 Bit 2 --> Control-Taste
 Bit 3 --> Alternate-Taste
           (Beispiel bei ON MENU KEY GOSUB)
```

MENU(14) Information über die gedrückte Taste. Im unteren Byte steht der ASCII-Code der gedrückten Taste, im höheren Byte der Scan-Code (Beispiel bei ON MENU KEY GOSUB).

MENU(15) Anzahl der Mausklicks, die zum Ereignis führten.

ON MENU BUTTON klicks,taste,zustand **GOSUB** proc

```
klicks,taste,zustand: iexp
proc: Name einer Prozedur
```

Dieser Befehl dient der Überwachung der Maustasten. In den Variablen klicks, taste und zustand wird definiert, welcher Maustastenzustand zu einer Verzweigung in die Prozedur proc führen soll.

In klicks steht die maximale Anzahl der zu registrierenden Maustastenklicks. In taste steht, welche Tastenkombination erwartet wird. Dabei gilt:

```
0 --> Eine Taste.
1 --> Linke taste.
2 --> Rechte Taste.
3 --> Beide Tasten gleichzeitig.
```

Der Ausdruck zustand gibt dabei an, welchen Zustand die betreffenden Tasten haben sollen. Dabei steht 0 für eine nicht gedrückte Taste und 1 für eine gedrückte Taste. Der Name proc gibt an, zu welcher Prozedur verzweigt werden soll, wenn das durch klicks, taste und zustand beschriebene Ereignis eingetreten ist.

Die Überwachung findet bei jedem ON MENU-Befehl statt.

Beispiel:

```
ON MENU BUTTON 1,1,0 GOSUB box
GRAPHMODE 3
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE box
  ADD i%,7
  IF i%>200
    i%=3
  ENDIF
  BOX 320-i%,200-i%,320+i%,200+i%
RETURN
```

--> Zeichnet eine grafische Spielerei auf den Bildschirm, solange die linke Maustaste nicht gedrückt wird. Das Drücken der rechten Maustaste beendet das Programm.

ON MENU KEY GOSUB proc

proc: Name einer Prozedur

Dieser Befehl richtet eine Überwachung der Tastatur ein. proc ist der Name einer Prozedur, zu der verzweigt wird, wenn eine Taste während eines ON MENU-Befehls gedrückt wurde.

Beispiel:

```
ON MENU KEY GOSUB key_auswertung
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE key_auswertung
  PRINT "Tastaturumschalttasten: ";MENU(13)
  PRINT "ASCII-Code: ";BYTE(MENU(14))
  PRINT "Scan-Code: ";SHR(MENU(14),8)
  PRINT
RETURN
```

--> Beim Drücken einer Taste wird der aktuelle Zustand der Tastaturumschalttasten (Shift, Control, Alternate) gemeldet sowie der ASCII- und Scan-Code der gedrückten Taste. Das Drücken der rechten Maustaste beendet das Programm. Zur Bedeutung von MENU(13) und MENU(14) siehe bei MENU(x).

ON MENU IBOX nr,x,y,b,h **GOSUB** proc
ON MENU OBOX nr,x,y,b,h **GOSUB** proc

```
nr,x,y,b,h: iexp
proc: Name einer Prozedur
```

Diese beiden Befehle überwachen die Mauskoordinaten. Wenn die Maus einen rechteckigen Bildschirmbereich betritt (IBOX) oder verläßt (OBOX), wird zur Prozedur proc verzweigt.

Dabei ist es möglich, zwei rechteckige Bildschirmbereiche zu definieren, die unabhängig voneinander überwacht werden. nr ist die Nummer des entsprechenden Rechtecks, x dessen linke X-Koordinate, y die obere Y-Koordinate, b die Breite und h die Höhe des Rechtecks.

Die Überwachung findet während der Ausführung eines ON MENU-Befehls statt.

Beispiel:

```
ON MENU IBOX 1,250,130,140,140 GOSUB rein_in_box
ON MENU OBOX 2,50,50,540,300 GOSUB raus_aus_box
'
GRAPHMODE 3
BOX 250,130,390,270
BOX 50,50,590,350
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE rein_in_box
  BOX 250+i%,130+i%,390-i%,270-i%
  IF i%=70
    i%=2
  ENDIF
  ADD i%,2
RETURN
'
PROCEDURE raus_aus_box
  BOX 0+j%,0+j%,639-j%,399-j%
  IF j%=0
    j%=50
  ENDIF
  SUB j%,2
RETURN
```

--> Betritt die Maus das innere Rechteck, findet dort eine Grafikspielerei statt. Verläßt sie das äußere Rechteck, so findet diese Spielerei dort statt. Das Drücken der rechten Maustaste beendet das Programm.

ON MENU MESSAGE GOSUB proc

`proc: Name einer Prozedur`

Wenn eine Nachricht im Message-Buffer eintrifft, so verzweigt das Programm in die Prozedur mit dem Namen proc. Die Überwachung des Message-Buffers findet bei jedem ON-MENU-Befehl statt. Der Aufbau des Message-Buffers wird im Abschnitt über MENU(x) besprochen.

Beispiel:

```
DIM m$(10)
FOR i%=0 TO 10
  READ m$(i%)
NEXT i%
DATA Desk, Redraw ,-------------------------
DATA 1,2,3,4,5,6,"","
OPENW 4,0,0
MENU m$()
ON MENU MESSAGE GOSUB message_auswertung
PRINT AT(1,1);
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE message_auswertung
  IF MENU(1)=20
    PRINT CHR$(7)
    PRINT "Ein Bildschirmausschnitt"
    PRINT "muß neu gezeichnet werden."
  ELSE
```

```
    PRINT CHR$(7)
    PRINT "Irgendetwas ist passiert !!!"
  ENDIF
RETURN
```

--> Zum Austesten dieses Listings muß ein Accessory geladen sein. Wird dieses angewählt, so merkt das Programm, daß ein Bildschirmausschnitt überdeckt wurde und meldet dies (siehe dazu auch MENU(x)). Zu Beginn des Programms erscheint diese Meldung ebenfalls einmal. Das Programm kann durch das Drücken der rechten Maustaste beendet werden.

Pull-Down-Menüs

In diesem Abschnitt werden die für GFA-BASIC 3.0 spezifischen Befehle zur Verwaltung von Pull-Down-Menüs eingeführt. Leider herrscht in der Literatur zu diesem Thema eine gewisse Begriffsverwirrung. Daher muß hier zunächst angegeben werden, welche Bedeutungen die damit verbundenen Begriffe in diesem Handbuch haben sollen. Den Begriff Pull-Down-Menü benutzen wir als Oberbegriff. In der obersten Bildschirmzeile befindet sich der ständig sichtbare Teil des Pull-Down-Menüs, die Menüleiste. Diese wiederum enthält die einzelnen Überschriften. Gelangt der Mauspfeil auf eine dieser Überschriften, so klappt darunter ein sogenanntes Menü auf. Jeder Teil dieses Menüs, der einzeln angewählt werden kann, wird als Menüeintrag bezeichnet. Diese Begriffswahl ist nicht allgemein verbindlich, sondern gilt nur für dieses Handbuch; in manchen Büchern werden andere Definitionen verwendet.

Bei der Einrichtung eines Pull-Down-Menüs werden dessen Einträge in einem String-Array m$() festgelegt. Mit dem Befehl MENU m$() wird dieses Pull-Down-Menü auf den Bildschirm gebracht. Der Befehl ON MENU GOSUB legt eine Prozedur fest, zu der nach der Wahl eines Menüeintrags verzweigt wird. Während des Programmlaufs wird bei jedem Auftreten eines ON MENU-Befehls geprüft, ob ein Eintrag gewählt wurde.

Der Befehl MENU OFF stellt invertierte Einträge der Menüleiste wieder in normaler Schrift dar. Mit MENU KILL wird das Pull-Down-Menü abgeschaltet.

Der Befehl MENU x,y erlaubt es, Menüeinträge mit Häkchen zu versehen oder in heller Schrift darzustellen, was dazu führt, daß dieser Menüeintrag nicht mehr anwählbar ist.

ON MENU GOSUB proc
MENU m$()

```
proc: Name einer Prozedur
m$(): String-Array
```

Diese beiden Befehle sind für die Erzeugung und Verwaltung eines Pull-Down-Menüs zuständig, wobei sie von den Befehlen und Variablen des vorigen Abschnitts unterstützt werden (ON MENU, MENU()). Mit ON MENU GOSUB proc wird eine Prozedur festgelegt, zu der nach der Anwahl eines Menüeintrags verzweigt wird. Handelt es sich bei diesem Eintrag um ein Accessory, so wird diese Prozedur nicht angesprungen. Innerhalb der Prozedur kann mit Hife der Variable MENU(0) herausgefunden werden, welcher Menüeintrag gewählt wurde. MENU(0) ist der Index des gewählten Eintrags im Array der Einträge m$(); m$(MENU(0)) ist also der im Pull-Down-Menü angeklickte Text, wenn OPTION BASE 0 gilt. Ist OPTION BASE 1 eingestellt, so ist der Text des gewählten Eintrags m$(MENU(0) + 1).

MENU m$() stellt das Pull-Down-Menü auf dem Bildschirm dar. Im Zeichenkettenfeld m$() stehen die Überschriften, Einträge und Platzhalter für die Accessories. Folgendes Format muß bei der Anordnung der Einträge im Feld m$ beachtet werden:

```
m$(0)            Überschrift des Menüs, in dem Accessories sein können.
m$(1)            Name des ersten Eintrags im ersten Menü.
m$(2)            Eine Folge von Minus-Zeichen (-).
m$(3) - m$(8)    Platzhalter für die Accessories.
                 Dabei handelt es sich lediglich um sechs Strings mit
                 einer Länge größer Null. Sind Accessories beim
                 Einschalten des Rechners geladen worden, so
                 erscheinen sie im Pull-Down-Menü unter ihrem Namen,
                 andernfalls werden die entsprechenden Menüeinträge
                 nicht dargestellt.
m$(9)            Ein Leer-String, der das Ende des ersten Menüs markiert.
```

Alle weiteren Menüs haben folgendes Format:

1. Überschrift des Menüs.
2. Liste der Menüeinträge.
3. Ein Leer-String, der das Ende des Menüs markiert.

Hinter dem letzten Menü folgt noch ein weiterer Leer-String, der das Ende des gesamten Pull-Down-Menüs markiert. Ein Menüeintrag, der mit einem Minus-Zeichen beginnt, wird automatisch hell dargestellt und ist nicht anwählbar.

Beispiel: Siehe Ende des Abschnitts.

MENU OFF
MENU KILL

Bei der Wahl eines Menüeintrags wird die Überschrift dieses Menüs invertiert. MENU OFF schaltet die Überschrift wieder in den Normalzustand.

MENU KILL schaltet das Pull-Down-Menü ab; es wird allerdings nicht vom Bildschirm entfernt. Außerdem schaltet MENU KILL die ON MENU GOSUB Einstellungen ab.

MENU x,y

```
x,y: aexp
```

Mit diesem Befehl kann der x-te Eintrag eines Pull-Down-Menüs umgestellt werden. Die Numerierung der Einträge entspricht den Indizes des String-Arrays, in dem die Pull-Down-Menüeintrage stehen. Die Zählung beginnt also bei Null; Überschriften, Dummy-Einträge für die Accessories und Leer-Strings werden mitgezählt.

Der zweite Parameter y gibt an, was mit dem x-ten Menüeintrag getan werden soll. Dabei gilt folgende Zuordnung:

```
y        Effekt

0  -->   Kein Häkchen davor.
1  -->   Häkchen davor setzen.
2  -->   Helle Schrift, nicht wählbar.
3  -->   Normale Schrift, wählbarer Eintrag.
```

Beispiel: Siehe Ende des Abschnitts.

Beispielprogramm für den Abschnitt Pull-Down-Menü:

```
DIM eintrag$(20)
i%=-1
REPEAT
  INC i%
  READ eintrag$(i%)
UNTIL eintrag$(i%)="Schlußmarke"
eintrag$(i%)=""
'
MENU eintrag$()
ON MENU GOSUB auswertung
OPENW 0
'
DATA Desk, Test ,--------------------,1,2,3,4,5,6,
DATA File, Load , Save ,------, Quit ,
DATA Titel, Eintrag 1 , Eintrag 2 ,
DATA Schlußmarke
'
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE auswertung
  MENU OFF
  ' MENU(0) enthält den Array-Index des angewählten Eintrags
  m%=MENU(0)
  PRINT eintrag$(m%)
  '
  ALERT 0,"Häckchen davor ?",1," Ja | Nein ",a%
  IF a%=1
    MENU m%,1
  ELSE
    MENU m%,0
  ENDIF
  '
```

```
  ALERT 0,"Helle Schrift ?|(nicht wählbar)",1," Ja | Nein ",a%
  IF a%=1
    MENU m%,2
  ELSE
    MENU m%,3
  ENDIF
RETURN
```

--> Erzeugt und verwaltet ein Pull-Down-Menü. Beim Anklicken eines Menüeintrags wird dessen Text auf den Monitor geschrieben und gefragt, wie der gewählte Menüeintrag dargestellt werden soll.

Fensterbefehle

GFA-BASIC bietet einige einfache Befehle zur Fensterverwaltung, die jedoch nicht sehr flexibel sind (OPENW, CLOSEW, CLEARW, TITLEW, INFOW). Will man eine leistungsfähigere Fensterverwaltung programmieren, muß man entweder die entsprechenden AES-Routinen benutzen oder die Parameter des gewünschten Fensters in einer Tabelle (WINDTAB) ablegen und das Fenster erneut aufrufen. Zusätzlich stehen die Funktionen W_HAND und W_INDEX als Bindeglied zwischen den einfachen GFA-BASIC-Befehlen zur Fensterverwaltung und den umfangreichen AES-Funktionen aus der Fensterbibliothek zur Verfügung.

Die einfachen Befehle gehen von wenig variablen Fensterpositionen aus. Eine Ecke des Fensters liegt immer an derselben Position, während die andere Ecke an der Position am Berührungspunkt der vier Fenster liegt.

Bei der Verwendung der einfachen Befehle wird der Nullpunkt des Koordinatensystems für Text- und Grafikausgaben auf die linke obere Fensterecke gelegt. Dies wird allerdings von einigen Befehlen, z.B. von GET und PUT, BITBLT usw., nicht beachtet. Grafik- und Textausgaben, die den Fensterbereich verlassen, werden automatisch abgeschnitten (Clipping).

OPENW #n,x,y,w,h,attr
OPENW nr [,x_pos,y_pos]
CLOSEW [#] nr

```
nr,x_pos,Y_pos: aexp
n,x,y,w,h,attr: iexp
```

Mit OPENW wird das Fenster mit der Nummer nr geöffnet. Die Parameter x_pos und y_pos legen die Position einer Fensterecke fest; die jeweils diagonal gegenüberliegende Ecke ist fest. Für eine variablere Fensterverwaltung benötigt man die AES-Routinen oder WINDTAB. Die Koordinaten (x,y) der möglichen Fenster sind:

Nr	Links oben	Rechts unten
1	(0,19)	(x_pos,y_pos)
2	(x_pos,19)	(639,y_pos)
3	(0,y_pos)	(x_pos,399)
4	(x_pos,y_pos)	(639,399)

Der Punkt (x_pos,y_pos) ist somit der Berührungspunkt der vier Fenster.

Mit dem Befehl OPENW 0 wird kein echtes Fenster geöffnet, sondern lediglich der Koordinatenursprung in den Punkt (0,19) verlegt. Dadurch sind die obersten 19 Zeilen des Bidschirms mit Grafik- oder Textausgabebefehlen nicht mehr zu erreichen. Dies ist z.B. sinnvoll, um eine Pull-Down-Menüleiste vor dem Überschreiben zu schützen.

Der Befehl CLOSEW schließt das Fenster mit Nummer nr, CLOSEW# das Fenster mit dem Index n.

Beispiel: (siehe auch Beispielprogramm am Ende des Abschnitts):

```
REPEAT
  IF MOUSEK=1
    CLOSEW 1
    OPENW 4,320,200
  ENDIF
  IF MOUSEK=2
    CLOSEW 4
    OPENW 1,100,100
  ENDIF
UNTIL MOUSEK=3
CLOSEW #1
CLOSEW #4
```

--> Das Drücken der linken Maustaste öffnet Fenster 1, das Drücken der rechten Maustaste Fenster 4. Gleichzeitiges Bedienen beider Maustasten beendet das Programm.

Die zweite Variante von OPENW öffnet das Fenster mit dem Index n an der Position x, y mit der Breite w und der Höhe h. Der Ausdruck attr legt fest, welche Elemente das Fenster haben soll (vergleiche WIND_CREATE im Abschnitt über die AES).

Beispiel:

```
TITLEW #1,"Titel"                        ! öffnet Fenster mit Index 1
INFOW # 1,STRING$(15,"....|")            ! Infozeile vorbelegen
OPENW #1,16,32,600,300,&X111111111111    ! Koordinaten + Attribute setzen
~INP(2)
CLOSEW #1                                ! wichtig!!, Fenster schließen
```

--> Öffnet ein Fenster mit Titel- und Infozeile. Ein Tastendruck beendet das Programm.

W_HAND(#n)
W_INDEX(#hd)

n, hd: aexp

W_HAND liefert das GEM-Handle des Fensters, dessen Kennung in n angegeben wurde. W_INDEX bildet dazu die Umkehrfunktion und meldet die Fensternummer zum anzugebenden GEM-Handle zurück.

Beispiel:

```
OPENW 2
PRINT W_HAND(#2)
~INP(2)
CLOSEW #2
```

--> Schreibt das Handle des Fensters mit der Kennung 2 auf den Bildschirm. Ein Tastendruck beendet das Programm.

CLEARW [#] nr
TITLEW [#] nr,a$
INFOW [#] nr,a$
TOPW #nr
FULLW [#] nr

Der Befehl CLEARW löscht das Fenster mit der Nummer nr. TITELW schreibt in die oberste Zeile des Fensters nr den Text a$. INFOW schreibt in die darunterliegende Informationszeile des Fensters nr den Text a$. TOPW aktiviert das Fenster mit der Nummer nr. FULLW bringt das Fenster nr auf volle Bildschirmgröße.

Beispiel: (siehe auch Beispielprogramm am Ende des Abschnitts):

```
DEFFILL 1,2,4
PBOX 0,0,639,399
OPENW 1
PAUSE 50
FULLW #1
PRINT "Fenster 1"
OPENW 4,100,100
PAUSE 50
CLEARW 1
OPENW 3
PAUSE 50
TOPW #1
PAUSE 50
CLOSEW #1
TITLEW 4,"Fenster 4"
INFOW 3,"Fenster 3"
PAUSE 100
CLOSEW #3
CLOSEW 4
```

--> Läßt einige Fenster erscheinen, verändert sie und baut sie wieder ab.

WINDTAB
WINDTAB(i,j)

i,j: iexp

Ab der in WINDTAB stehenden Adresse steht die Adresse der Fenster-Parameter-Tabelle, in der sich alle Informationen, die das Aussehen eines Fensters bestimmen, befinden. Darüber hinaus kann in dieser Tabelle der Nullpunkt für Grafikausgaben festgelegt werden.

Die Tabelle besteht aus 68 Bytes und ist wortweise aufgebaut (2 Bytes sind jeweils ein Tabelleneintrag). Die Benutzung der Tabelle wird am Ende des Abschnitts in einem Beispielprogramm erläutert. In diesem Programm werden die Parameter des aufzubauenden Fensters in die Fenster-Tabelle eingetragen und das Fenster dann mit OPENW geöffnet.

Dabei kann WINDTAB (ähnlich INTIN()) auch als zweidimensionales Feld WINDTAB() angesprochen werden.Der erste Index ist die Nummer des Fensters (1 bis 4 oder 0), der zweite Index ist:

0 für das Handle, 1 für die Attribute, 2 für die x-, 3 für die y-Koordinate, 4 für die Breite und 5 für die Höhe des Fensters (Außenmaße).

WINDTAB kann auch über DPOKE angesprochen werden, die einzelnen Offsets haben folgende Bedeutung:

Offset	Funktion
0	Kennung (Handle) von Fenster 1.
2	Attribute für Fenster 1 (Aufbau siehe unten).
4	X-Koordinate für Fenster 1.
6	Y-Koordinate für Fenster 1.
8	Breite für Fenster 1.
10	Höhe für Fenster 1.
12 bis 22	Entsprechende Werte für Fenster 2.
24 bis 34	Entsprechende Werte für Fenster 3.
36 bis 46	Entsprechende Werte für Fenster 4.

48	-1
50	0
52 bis 58	Koordinaten und Größe für das Desktop-Fenster (0)
60 und 62	Koordinaten des Fensterberührungspunktes.
64 und 65	Nullpunkt für Grafikbefehle (CLIP OFFSET).

Der Nullpunkt wird von allen AES-, Line-A- und direkten VDI-Aufrufen sowie von PUT, GET und BITBLT nicht beachtet.

Der Aufbau des Fensterattributeintrags ist bitweise aufgebaut. Jedes Bit steht für einen Bestandteil des Fensters. Ist ein Bit gesetzt, so besitzt das Fenster das dazugehörende Fensterelement:

Bit	Zugehöriges Fensterelement
0	Fenstertiteleintrag.
1	Schließfeld links oben.
2	Voll-Feld rechts oben.
3	Kopfzeile, mit der das Fenster verschoben werden kann.
4	Info-Zeile unter der Titelzeile.
5	Größenveränderungsfeld rechts unten.
6	Nach oben zeigender Pfeil.
7	Nach unten zeigender Pfeil.
8	Senkrechter Schieberegler rechts.
9	Nach links zeigender Pfeil.
10	Nach rechts zeigender Pfeil.
11	Waagerechter Schieberegler unten.

Beispiel:

```
' Es ist auch möglich, den OPENW #n,x,y,w,h,attr-Befehl über WINDTAB zu
' simulieren, was in der Version 2.0 allein möglich war
'
OPENW #1,100,120,200,70,&HFFF
'
' entspricht
'
DPOKE WINDTAB+2,&HFFF
DPOKE WINDTAB+4,100
DPOKE WINDTAB+6,120
DPOKE WINDTAB+8,200
DPOKE WINDTAB+10,70
OPENW 1
'
' oder
'
WINDTAB(1,1)=&HFFF
WINDTAB(1,2)=100
WINDTAB(1,3)=120
WINDTAB(1,4)=200
WINDTAB(1,5)=70
OPENW 1
```

Sonstige

RC_INTERSECT(x1,y1,w1,h1,x2,y2,w2,h2)

```
x1,y1,w1,h1 : iexp
x2,y2,w2,h2 : ivar
```

Die Funktion RC_INTERSECT (rectangle intersection) dient zur Feststellung, ob sich zwei Rechtecke (gegeben durch die Koordinaten des linken oberen Eckpunktes (x,y) sowie Breite w und Höhe h) überlappen.

Schneiden sich die beiden Rechtecke, so wird TRUE zurückgegeben und x2,y2,w2,h2 enthalten nach dem Funktionsaufruf die Koordinaten der Schnittfläche.

Ergibt sich jedoch keine Schnittfläche, wird FALSE zurückgegeben und x2,y2,w2,h2 enthalten die Koordinaten eines Rechteckes, das zwischen den beiden angegebenen Rechtecken liegt. Die Breite w2 oder die Höhe h2 sind dann negativ oder gleich Null.

Angewendet wird diese Funktion normalerweise zur Behandlung des 'Redraws' bei GEM-Events.

Beispiel:

```
BOX 100,100,400,300
x=200
y=200
w=300
h=150
BOX x,y,x+w,y+h
'
IF RC_INTERSECT(100,100,300,200,x,y,w,h)
  PBOX x,y,x+w,y+h
ENDIF
```

--> Es werden zwei Rechtecke gezeichnet und die Schnittfläche schwarz dargestellt.

RC_COPY s_adr,sx,sy,w,h **TO** d_adr,dx,dy [,m]

`s_adr,d_adr,sx,sy,w,h,dx,dy,m: iexp`

Der Befehl RC_COPY ermöglicht das Kopieren von Bildrechtecken zwischen mehreren Bildseiten. Die Parameter s_adr und d_adr enthalten die Adresse des Quell- (source) und Zielbildschirms (destination). Die Koordinaten der linken oberen Ecke sowie Breite und Höhe des zu kopierenden Rechteckes werden in sx,sy, w und h übergeben. Die Koordinaten der obereren linken Ecke des Zielrechteckes sind dx und dy. Wahlweise kann ein Verknüpfungsmodus (0 bis 15, vgl. PUT) angegeben werden. Voreingestellt der Wert 3 für m verwendet.

Beispiel:

```
FILESELECT "\*.*","",f$
IF EXIST(f$)
  OPEN "i",#1,f$
  bild$=INPUT$(32000,#1)
  s_adr%=V:bild$
  d_adr%=XBIOS(2)
  '
  FOR i%=1 TO 1000
    RC_COPY s_adr%,RAND(10)*64,RAND(10)*40,64,40 TO
d_adr%,RAND(10)*64,RAND(10)*40
  NEXT i%
  '
  CLOSE #1
ENDIF
```

--> Eine Datei im Bildschirmformat (32000 Byte) wird geladen und in zufälligen Ausschnitten auf dem Bildschirm dargestellt.

ALERT sym,text$,default,button$,wahl

```
sym,default: iexp
text$,button$: sexp
wahl: avar
```

Der Befehl ALERT läßt eine Alert-Box auf dem Bildschirm erscheinen. Im Ausdruck sym wird festgelegt, welches Symbol in der linken Hälfte der Alert-Box stehen soll. Es gilt:

```
0  -->  Kein Symbol.
1  -->  Ausrufezeichen.
2  -->  Fragezeichen.
3  -->  Stop-Schild.
```

Im String-Ausdruck text$ steht der Text, der in der Alert-Box erscheinen soll. Es sind maximal 4 Zeilen mit jeweils höchstens 30 Zeichen erlaubt. Die Zeilen werden durch ein | voneinander getrennt. Zeilen, die länger als 30 Zeichen sind, werden abgeschnitten.

Der Ausdruck default gibt an, der wievielte Auswahlknopf der Alert-Box der sogenannte Default-Button ist. Dieser Auswahlknopf kann nicht nur mit der Maus, sondern auch durch Drücken der Return- oder Enter-Taste gewählt werden. Existiert kein Button mit der Nummer default, so kann die Alert-Box nicht mit Return oder Enter verlassen werden, sondern nur mit der Maus.

Der String-Ausdruck button$ enthält den Text der Menübuttons. Die maximale Textlänge pro Button ist 8 Zeichen. Die Texte der einzelnen Buttons werden durch | getrennt.

In der Variable wahl wird die Nummer des gewählten Buttons zurückgemeldet (vgl. FORM_ALERT).

Beispiel:

```
ALERT 1,"Wählen Sie|einen Button !",1,"Links|Rechts",a%
ALERT 0,"Sie haben Button "+STR$(a%)+" gewählt.",0," Ok ",a%
```

--> Es erscheint eine Alert-Box mit zwei Buttons. Nach der Wahl eines Buttons erscheint eine zweite Box, die angibt, welcher Button in der ersten Box gewählt wurde. Diese zweite Alert-Box besitzt kein Symbol und keinen Default-Button.

FILESELECT pfad$,default$,name$

```
pfad$,default$: sexp
name$: svar
```

Dieser Befehl läßt eine File-Select-Box auf dem Bildschirm erscheinen, mit deren Hilfe aus dem Inhaltsverzeichnis einer Diskette (RAM-Disk, Festplatte, usw.) eine Datei ausgewählt werden kann.

Im Ausdruck pfad$ steht der Name des Laufwerks und des Ordners, dessen Inhaltsverzeichnis aufgeführt werden soll. Ist kein Laufwerk angegeben, so wird das aktuelle Laufwerk gewählt. default$ ist der Name der Datei, die links in der File-Select-Box voreingestellt erscheinen soll. In name$ findet sich der Name der gewählten Datei, nachdem man den Ok-Button der File-Select-Box angeklickt hat. Wurde der Abbruch-Button gewählt, so enthält name$ einen Leer-String.

Das Format von pfad$, default$ und name$ entspricht den Konventionen des hierarchischen Dateisystems, das im Kapitel "Allgemeine Ein- und Ausgabe", Abschnitt "Dateiverwaltung", beschrieben wird.

(Vergleiche FSEL_INPUT.)

Beispiel:

```
FILESELECT "A:\*.PRG","GFABASIC.PRG",name$
IF name$=""
  PRINT "Sie haben den Abbruch-Knopf gewählt."
ELSE IF RIGHT$(name$)="\"
  PRINT "Sie haben den OK-Knopf ohne Dateiangabe gewählt."
ELSE
  PRINT "Sie haben die Datei: ";name$;" gewählt."
ENDIF
```

--> Läßt eine File-Select-Box erscheinen und analysiert die vom Benutzer getroffene Wahl.

10 - Systemroutinen

GEMDOS, BIOS und XBIOS

GEMDOS(nr [,x,y...])
BIOS(nr [,x,y...])
XBIOS(nr [,x,y...])

nr,x,y: iexp

Diese drei Funktionen erlauben das Aufrufen von GEMDOS-, BIOS- und XBIOS-Routinen. Zu diesem Zweck werden jeweils die Funktionsnummer nr und die Parameterliste übergeben. Um die Parameter im richtigen Variablenformat übergeben zu können, kann vor den Parametern ein W: oder ein L: gesetzt werden. Die mit W: gekennzeichneten Parameter und die ohne Kennung werden dann als Worte (16 Bit), die mit L: versehenen Werte als Langworte (32 Bit) übergeben.

Siehe hierzu auch Anhang.

Beispiele:

```
IF GEMDOS(17)
  PRINT "Drucker bereit"
ELSE
  PRINT "Drucker nicht bereit"
ENDIF
```

--> Prüft, ob ein Drucker an der parallelen Schnittstelle empfangsbereit ist (GEMDOS(17) meldet den Ausgabestatus der parallelen Schnittstelle zurück).

```
REPEAT
UNTIL BIOS(11,-1) AND 4
```

--> Wartet, bis die Control-Taste gedrückt wird (BIOS(11,-1) meldet den Status der Tastaturumschalttasten zurück).

```
CIRCLE 320,200,180
BMOVE XBIOS(2),XBIOS(2)+16000,16000
```

--> Zeichnet einen Kreis, kopiert dessen obere Hälfte in die untere Bildschirmhälfte (XBIOS(2) meldet die Adresse zurück, an der der physikalische Bildschirmspeicher beginnt).

L:x
W:x

x: iexp

Diese beiden Funktionen dienen der Übergabe von numerischen Ausdrücken als Wort (2 Byte, W:) oder Langwort (4 Byte, L:) beim Aufruf von Betriebssystemfunktionen und C-Routinen. Ohne Kennung wird das Wortformat benutzt.

Beispiel:

```
CLS
FOR i% = 1 TO 100
  BOX RAND(639),RAND(399),RAND(639),RAND(399)
NEXT i%
PBOX 0,0,639,399
DIM screen_2|(32255)
phys_base%=XBIOS(2)
old_screen%=phys_base%
log_base%=V:screen_2|(0)+255 AND &HFFFFFF00
REPEAT
  IF MOUSEK=1
    ~XBIOS(5,L:log_base%,L:phys_base%,-1)
    SWAP log_base%,phys_base%
  ENDIF
  PLOT MOUSEX,MOUSEY
UNTIL MOUSEK=2
~XBIOS(5,L:old_screen%,L:old_screen%,-1)
```

--> Schaltet beim Drücken der linken Maustaste zwischen zwei Bildschirmseiten um. Das Drücken der rechten Maustaste führt zum Programmabbruch (siehe auch XBIOS). Außerdem wird auf beiden Bildseiten an der Mausposition geplottet.

Die Anzahl und Bedeutung der Parameter ist, ebenso wie der Rückgabewert, von der aufgerufenen Betriebssystemroutine abhängig.

Line-A-Aufrufe

ACLIP, PSET, PTST, ALINE, HLINE, ARECT, APOLY, ACHAR, ATEXT

Im folgenden Abschnitt werden Befehle besprochen, die grundsätzlich Befehlen entsprechen, die schon im Kapitel Grafik vorgestellt wurden. Die Grafikausgaben über die Line-A-Aufrufe sind jedoch teilweise wesentlich schneller und benutzen eine etwas andere Syntax. Vor Line-A-Grafikausgaben sollte immer mit ACLIP ein Clipping gesetzt werden, da ansonsten Speicherbereiche außerhalb des Bildschirmspeichers überschrieben werden können. Sonst wird auch das Clipping eines beliebigen vorherigen Grafikbefehls (VDI oder AES) benutzt.

Ein VDI-Aufruf verändert das mit ACLIP eingestellte Clipping. Line-A-Aufrufe sind unabhängig von den VDI-DEFxxx-Anweisungen.

Zur Farbwahl: Die bei den Line-A-Routinen angegebenen Farben entsprechen den hardware-mäßigen Farbregisternummern (wie SETCOLOR) und nicht den Nummern, die das VDI verwendet (COLOR).

ACLIP flag,xmin,ymin,xmax,ymax

```
flag,xmin,ymin,xmax,ymax: iexp
```

Dieser Befehl ermöglicht die Festlegung eines Begrenzungsrechteckes, d.h. Grafikausgaben über Line-A werden auf diesen rechteckigen Bildschirmausschnitt begrenzt. In xmin,ymin, xmax und ymax stehen die Koordinaten der linken oberen und der rechten unteren Ecke des Clip-Rechtecks. Bei flag< >0 wird das Clipping ein-, sonst ausgeschaltet. ACLIP gilt (leider) nicht für PSET, PTST, ALINE, HLINE und BITBLT.

PSET x,y,f

```
x,y,f: iexp
```

PSET entspricht dem PLOT-Befehl und ermöglicht das Setzten von Punkten an den Koordinaten x,y in der Farbe f. Dabei kann f, je nach Auflösung, einen Wert von 0 bis 15 annehmen.

Beispiel:

```
FOR i%=0 TO 199 STEP 2
  PSET i%,i%,1
NEXT i%
```

--> Setzt Punkte mit einem Abstand von zwei Pixeln von 0,0 nach 199,199.

PTST(x,y)

`x,y: iexp`

Die Funktion PTST entspricht der POINT-Funktion. Sie liefert die Farbe zurück, die an der Bildschirmposition x,y vorliegt.

Beispiel:

```
PSET 100,100
x=PTST(200,100)
PRINT x,PTST(100,100)
```

--> Gibt die Farben an, die an den Bildschirmpositionen 200,100 und 100,100 vorliegen.

ALINE x1,y1,x2,y2,f,lm,m

```
x1,y1,x2,y2,f,lm,m: iexp
```

ALINE entspricht dem LINE-Befehl, dabei enthalten x1,y1,x2,y2 die Koordinaten des Anfangs- und Endpunktes der Linie. Der Ausdruck f enthält die Farbinformation und kann, abhängig von der jeweiligen Auflösung, Werte von 0 bis 15 annehmen (siehe COLOR). In lm wird die Bit-Information für das gewünschte Linienmuster übergeben. Dabei handelt es sich um eine Maske mit 16 Bit Länge, in der jedes gesetzte Bit (im Monochrom-Modus) einem zu zeichnenden Punkt entspricht.

Der Parameter m legt den Grafikmodus fest und kann Werte von 0 bis 3 annehmen. Dabei bedeutet:

```
0 --> Replace
1 --> Transparent
2 --> Invertieren
3 --> Invertiert transparent
```

Beispiel:

```
ym%=INT{L~A-4}-1
FOR i%=0 TO 199
  ALINE i%,0,i%,ym%,1,&HFFFF,0
NEXT i%
```

--> Zeichnet senkrechte Linien über die ganze Bildschirmhöhe von der x-Koordinate 0 bis 199.

HLINE x1,y,x2,f,m,adr,anz_muster

```
x1,x2,y,f,m,adr,anz_muster: iexp
```

HLINE entspricht dem LINE-Befehl, jedoch können mit diesem Befehl nur waagerechte Linien gezeichnet werden. Dabei enthalten x1 und x2 die beiden x-Koordinaten und y die y-Koordinate der zu zeichnenden Linie. Der Ausdruck f enthält die Farbinformation und kann, abhängig von der jeweiligen Auflösung, Werte von 0 bis 15 annehmen (siehe COLOR). Der Parameter m legt den Grafikmodus fest und entspricht dem bei ALINE.

Adr ist die Adresse eines Feldes, in dem die Bit-Informationen für die Linienmuster (je 16 Bit) stehen. Anz_muster ist eine Maske, mit der die Y-Koordinate verknüpft wird, um dann als Index in der Linienmustertabelle zu dienen. Deshalb sollte anz_muster um eins kleiner als eine Zweier-Potenz sein (0,1,3,7,15,...).

Beispiel:

```
ACLIP 1,0,0,639,399
'
muster%=&X1111111111111111010101010101010
z%=V:muster%
'
FOR i%=0 TO 199
  HLINE 0,i%,639,1,0,z%,1
NEXT i%
```

--> In der Variable muster% werden zwei 16-Bit-Linienmuster abgelegt. Der letzte HLINE-Parameter ist dementsprechend 1 (2 Muster). Die mit HLINE gezeichneten Linien benutzen dann abwechselnd die beiden 16-Bit-Linienmuster, die in muster% stehen.

ARECT x1,y1,x2,y2,f,m,adr,anz_muster

```
x1,y1,x2,y2,f,m,adr,anz_muster: iexp
```

ARECT entspricht PBOX. Dabei sind x1,y1 und x2,y2 zwei gegenüberliegende Eckpunkte des Rechtecks. Die Parameter f, m, adr und anz_muster haben die gleiche Bedeutung wie bei HLINE.

Beispiel:

```
ACLIP 1,0,0,639,399
DIM muster&(1)
'
muster&(0)=&X1010101010101010
muster&(1)=&X0101010101010101
'
muster_adr%=V:muster&(0)
ARECT 100,100,200,200,1,0,muster_adr%,1
```

--> Zeichnet ein gefülltes Rechteck mit einem Muster, das dem Hintergrund beim Desktop entspricht (d.h. DEFFILL 1,2,4).

APOLY adr_pkt,anz_pkt,y0 **TO** y1,f,m,adr,anz_muster

```
adr_pkt,anz_pkt,y0,y1,f,m,adr,anz_muster: iexp
```

APOLY hat Ähnlichkeit mit dem POLYFILL-Befehl, zeichnet einen unsichtbaren geschlossenen Linienzug mit n Ecken und füllt ihn mit einem beliebigen Muster. Dazu wird in adr_pkt die Adresse des Arrays übergeben, welches abwechselnd die x- und y-Koordinaten der Punkte enthält. Der Parameter 'anz_pkt' enthält die Anzahl der Punkte. In y0 und y1 werden die Y-Koordinaten angegeben, die bestimmen, welcher Teil des Polygons gefüllt werden soll. Die Parameter f, m, adr, anz_muster entsprechen denen bei HLINE.

Beispiel:

```
DIM x&(9),y&(9),muster&(1)
FOR i%=0 TO 8
  x&(i%)=RAND(100)
NEXT i%
x&(9)=x&(0)
'
adr_kor%=V:x&(0)
muster&(0)=-1
adr_muster%=V:muster&(0)
'
ACLIP 1,0,0,200,200
'
FOR i%=1 TO 8
  APOLY adr_xkor%,9,0 TO 100,1,1,adr_muster%,0
NEXT i%
```

--> Zeichnet einen geschlossenen Polygonzug, der anschließend gefüllt wird.

BITBLT adr%
BITBLT x%()

```
adr%: iexp
x%(): Vier-Byte-Integer-Array
```

Der Befehl BITBLT ruft die gleichnamige Line-A-Routine auf. Im Gegensatz dazu ruft der BITBLT-Befehl mit drei Parameter-Arrays eine VDI-Routine auf. In der Befehlsvariation, bei der eine Adresse als Parameter angegeben wird, muß ab dieser Adresse eine 76 Byte lange Tabelle stehen. Die Bedeutung der Werte in diesem Speicherbereich können Sie der nachfolgenden Tabelle entnehmen. Dabei steht in der Spalte Offset der Abstand der Tabellenelemente vom Tabellenbeginn in Bytes.

In der Befehlsvariation bei der ein Array aus Vier-Byte-Integern mit mindestens 23 Elementen angegeben werden muß, steht in jedem Array-Element ein Parameter. Welcher Parameter in welchem Array-Element stehen muß, können Sie der Spalte Index in der nachfolgenden Tabelle entnehmen.

Die mit einem Stern markierten Parameter werden von der BITBLT-Routine verändert. Daher sollte man normalerweise mit der Befehlsvariation arbeiten, die ein Array als Parameter benutzt, da von den dort stehenden Parametern eine Kopie erzeugt wird, auf die die BITBLT-Routine dann zugreift. Die Arrayelemente werden dabei nicht verändert. Im Gegensatz dazu werden die Tabellenelemente bei der Übergabe einer Adresse verändert.

Name		Index	Offset	Bedeutung
B_WD		00	00	Rasterbreite in Pixeln.
B_HT		01	02	Rasterhöhe in Pixeln.
PLANE_CT	*	02	04	Anzahl der Farb-Ebenen.
FG_COL	*	03	06	Vordergrundfarbe
BG_COL	*	04	08	Hintergrundfarbe
OP_TAB		05	10	Logische Verknüpfung für jede Kombination von Vorder- und Hintergrund-Pixel.
S_XMIN		06	14	x-Offset im Quellraster.
S_YMIN		07	16	y-Offset im Quellraster.
S_FORM		08	18	Adresse des Quellrasters.
S_NXWD		09	22	Offset zum nächsten Wort der gleichen Ebene.
S_NXLN		10	24	Offset zur nächsten Zeile des Quellrasters
S_NXPL		11	26	Offset zur nächsten Farbebene (2).
D_XMIN		12	28	x-Offset im Zielraster.
D_YMIN		13	30	y-Offset im Zielraster.
D_FORM		14	32	Adresse des Zielrasters.
D_XNWD		15	36	Offset zum nächsten Wort der gleichen Ebene.
D_NXLN		16	38	Offset zur nächsten Zeile des Zielrasters.
D_NXPL		17	40	Offset zur nächsten Farbebene (immer 2).
P_ADDR		18	42	Zeiger auf Tabelle mit Füllmuster, wenn nicht Null, so wird mit AND verknüpft.
P_NXLN		19	46	Offset zur nächsten Zeile des Maskenrasters.
P_NXPL		20	48	Offset zur nächsten Farbe des Füllmusters.
P_MASK		21	50	Maske wie bei HLINE,
SPACE	*	22	52	die folgenden 24 Bytes dienen als Arbeitsbereich für den Blitter.

Beispiele:

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEP 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6       !nur wenn auch etwas da ist
    xx%=V:x%(0)
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    INT{xx%}=1
    INT{xx%+2}=h%
    INT{xx%+4}=1
    INT{xx%+6}=1
    INT{xx%+8}=0
    {xx%+10}=&H3030303
    INT{xx%+14}=9999
    INT{xx%+16}=0
    {xx%+18}=a%+6
    INT{xx%+22}=2
    INT{xx%+24}=SHR(b%+16,4)*2
    INT{xx%+26}=2
    INT{xx%+28}=9999
    INT{xx%+30}=0
    {xx%+32}=XBIOS(3)
    INT{xx%+36}=2
    INT{xx%+38}=80
    INT{xx%+40}=2
    {xx%+42}=0     !pattadr
```

```
    INT{xx%+46}=0     !p_nxtln
    INT{xx%+48}=0     !p_nxpl
    INT{xx%+50}=0     !p_mask
    '
    ABSOLUTE i&,xx%+14
    ABSOLUTE di&,xx%+28
    '
      FOR i&=0 TO b%
        INT{xx%+4}=1
        di&=SUB(639,i&)
        BITBLT xx%
      NEXT i&
      '
  ENDIF
RETURN
```

--> Ein Beispiel für BITBLT adr%.

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEP 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6       !nur wenn auch etwas da ist
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    x%(0)=1
    x%(1)=h%
    x%(2)=1
    x%(3)=1
```

```
    x%(4)=0
    x%(5)=&H3030303
    x%(6)=9999
    x%(7)=0
    x%(8)=V:x$+6
    x%(9)=2
    x%(10)=SHR(b%+16,4)*2
    x%(11)=2
    x%(12)=9999
    x%(13)=0
    x%(14)=XBIOS(3)
    x%(15)=2
    x%(16)=80
    x%(17)=2
    x%(18)=0     !pattadr
    x%(19)=0     !p_nxtln
    x%(20)=0     !p_nxpl
    x%(21)=0     !p_mask
    '
    FOR i%=0 TO b%
      x%(6)=i%
      x%(12)=639-i%
      BITBLT x%()
    NEXT i%
    '
  ENDIF
RETURN
```

--> Das ist ein Beispiel zu BITBLT x%().

Beide Routinen spiegeln ein mit GET eingelesenen Ausschnitt auf dem Bildschirm. Wie auch das Listing zu BITBLT im Kapitel Allgemeine Grafikbefehle. In der Routine, die BITBLT adr% verwendet, muß die Anzahl der Bitplanes jedesmal wieder gesetzt werden. Welche der BITBLT-Routinen verwendet wird, ist mehr oder weniger Geschmackssache. Geschwindigkeitsmäßig gibt es wenig Unterschiede.

ACHAR code,x,y,font,stil,winkel

```
code,x,y,font,stil,winkel: iexp
```

Mit Hilfe von ACHAR kann ein einzelnes Zeichen mit dem beliebigen ASCII-Wert code am Punkt (x,y) ausgegeben werden. Dabei kann der numerische Ausdruck font Werte zwischen 0 und 2 annehmen. Diesen Werten sind folgende Zeichensätze zugeordnet:

```
0 = 6x6  (Icon-Schrift)
1 = 8x8  (normale Schrift im Farbmodus)
2 = 8x16 (normale Monochromschrift)
```

Größere Werte für font werden als Font-Header-Adressen aufgefaßt. Ein solcher Font muß in dem Format vorliegen, in das er vom GDOS beim Laden umgewandelt wird, das heißt im Motorola-Format mit High-Byte vor Low-Byte.

Zusätzlich können Textstil (0 bis 31) und Ausgabewinkel (0,900,1800,2700) festgelegt werden. Diese Werte entsprechen denen des TEXT-Befehls. Im Gegensatz zum TEXT-Befehl beziehen sich die Koordinatenangaben jedoch auf die Zeichenoberkante.

ATEXT x,y,font,s$

```
x,y, font: iexp
s$: sexp
```

Der Befehl ATEXT dient der Ausgabe von Zeichenketten an beliebigen Bildschirmpositionen. Anders als bei ACHAR kann aber kein Schriftstil oder ein bestimmter Ausgabewinkel angegeben werden. Die Parameter x, y und font sind die gleichen wie bei ACHAR.

Beispiel:

```
a_clock                          ! ruft die Prozedur a_clock
'                                  zum erstenmal auf
EVERY 400 GOSUB a_clock          ! alle 2 Sekunden a_clock aufrufen
OPENW 0                          ! oberste Zeile vor Überschreiben
'                                ! schützen
FOR i%=1 TO 100000
  PRINT USING " ###### ",i%;     ! sonst Ziffern ausgeben
NEXT i%
'
PROCEDURE a_clock
  ACLIP 1,20,0,120,15            ! Clipping einschalten, sonst gilt das
  '                                des OPENW 0, d.h.: Man sieht nichts.
  ATEXT 20,0,2,TIME$             ! Uhrzeit ausgeben
RETURN
```

--> Gibt alle 2 Sekunden die Uhrzeit aus und schreibt Zahlenkolonnen auf den Bildschirm.

L~A

Die Funktion L~A ermittelt die Basisadresse der Line-A-Variablen (siehe Anhang).

Beispiel:

```
PRINT INT{L~A}
```

--> Gibt die Anzahl der Bitplanes in der aktuellen Auflösung aus. Siehe auch die Beispiele unter BITBLT.

VDI-Aufrufe

Die VDI-Funktionen werden in sieben Bereiche unterteilt:

- Kontroll-Funktionen.
- Ausgabe-Funktionen.
- Attribut-Funktionen.
- Raster-Funktionen.
- Eingabe-Funktionen.
- Auskunfts-Funktionen.
- Escapes.

Es werden drei Arten von Parametern unterschieden: Eingabe, Ausgabe und solche, die sowohl für Ein- als auch für Ausgabe zuständig sind. Diese Parameter werden in fünf Arrays übergeben:

- CONTRL Kontrolle.
- INTIN Integer-Eingabe.
- PTSIN Punktkoordinaten-Eingabe.
- INTOUT Integer-Ausgabe.
- PTSOUT Punktkoordinaten-Ausgabe.

Die Eingabeparameter stehen in:

CONTRL(0) Opcode.

CONTRL(1) Anzahl der Punkte im PTSIN-Arrays.

CONTRL(3) Länge des INTIN-Arrays.

CONTRL(5) Identifikation des Opcodes.

INTIN() Eingabe-Array der Integer-Werte.

PTSIN() Eingabe-Array der Punktkoordinaten.

Die Ausgabeparameter sind:

CONTRL(2) Anzahl der Punkte im PTSOUT-Array.

CONTRL(4) Länge des INTOUT-Arrays.

INTOUT() Ausgabe-Array der Integer-Werte.

PTSOUT() Ausgabe-Array der Punktkoordinaten.

Ein- und Ausgabeparameter sind:

CONTRL(6) Gerätekennung.

CONTRL(7-n) Vom Opcode abhängige Informationen.

Die Werte, die beim Aufruf einer VDI-Funktion in die entsprechenden Felder eingetragen werden müssen, sind je nach Funktion verschieden.

CONTRL
INTIN
PTSIN
INTOUT
PTSOUT

Diese Funktionen ermitteln die Adressen der VDI-Parameterblöcke. Mit einem Index in Klammern werden Elemente dieser Felder angesprochen. Die Adressen sind:

```
CONTRL --> Adresse des VDI-Control-Feldes.
INTIN  --> Adresse des VDI-Integer-Input-Feldes.
PTSIN  --> Adresse des VDI-Punktkoordinaten-Input-Feldes.
INTOUT --> Adresse des VDI-Integer-Output-Feldes.
PTSOUT --> Adresse des VDI-Punktkoordinaten-Output-Feldes.
```

Diese Felder enthalten die Parameter für VDI-Aufrufe.

Zum Beispiel entspricht CONTRL(2) dem DPOKE CONTRL+4 bzw. DPEEK(CONTRL+4). Auch die anderen Felder sind wortweise organisiert.

VDISYS [opc [,c_int,c_pts [,subopc]]]

```
opc,c_int,c_pts,subopc: iexp
```

Mit dem Befehl VDISYS wird die VDI-Funktion mit der Funktionsnummer opc aufgerufen. Wird opc nicht angegeben, so muß die Funktionsnummer, ebenso wie die anderen Parameter, mit DPOKE CONTRL,opc oder CONTRL(0) = opc in das Control-Feld eingetragen werden.

In den Parametern c_int und c_pts kann die Anzahl der Werte im Integer-Input-Feld und im Punktkoordinaten-Input-Feld angegeben werden. Diese Werte müssen dann nicht mehr in das Control-Feld eingetragen werden. Der optionale Parameter subopc enthält den Subopcode der aufzurufenden VDI-Routine. Dieser muß nur bei einigen VDI-Routinen mit angegeben werden, z.B. bei den Escape-Routinen. Die Parameter c_int, c_pts und subopc werden also in bestimmte CONTRL-Feld-Einträge eingesetzt.

Es gilt:

```
opc    --> CONTRL(0)
c_int  --> CONTRL(3)
c_pts  --> CONTRL(1)
subopc --> CONTRL(5)
```

Beispiele:

```
CONTRL(1)=3
CONTRL(5)=4
PTSIN(0)=320
PTSIN(1)=200
PTSIN(4)=190
VDISYS 11

PTSIN(0)=320
PTSIN(1)=200
```

```
PTSIN(4)=190
VDISYS 11,0,3,4

PCIRCLE 320,200,190
```

--> Zeichnet dreimal einen gefüllten Kreis auf den Bildschirm.

```
VDISYS 5,0,0,13
PRINT " Invers "
VDISYS 5,0,0,14
PRINT " Normal "
```

--> Schreibt den Text 'Invers' invertiert und den Text 'Normal' nicht invers.

VDIBASE

Die Systemvariable VDIBASE enthält die Adresse, ab der die aktuelle GEM-Version Parameter für die Verwaltung der VDI-Routinen ablegt. Der Aufbau dieses Speicherbereichs kann bei zukünftigen GEM-Versionen anders sein als bei der jetzigen. Deshalb entfällt ein Beispiel an dieser Stelle. VDIBASE ist im GFA-BASIC enthalten, um Programmierern die Möglichkeit zu geben, an alle VDI-Parameter heranzukommen.

WORK_OUT(x)

x: iexp

Diese Funktion ermittelt die Werte, die beim Aufruf der VDI-Funktion OPEN WORKSTATION in intout(0) bis intout(44) und ptsout(0) bis ptsout(1) stehen. Nur läuft der Index bei Verwendung der WORK_OUT-Funktionen von 0 bis 56 (siehe Anhang).

Beispiel:

```
raster_breite&=WORK_OUT(0)
raster_hoehe&=WORK_OUT(1)
PRINT raster_breite&,raster_hoehe&,WORK_OUT(10)
```

--> Gibt die Zahlen 639 und 399 für die Rasterbreite und -höhe (im Monochrom-Modus) und eine 1 für die Anzahl der verfügbaren Zeichensätze zurück (WORK_OUT(10).

Spezielle VDI-Routinen und GDOS

Die folgenden VDI-Workstation- und Abfragefunktionen sind nur verfügbar, wenn beim Booten das GDOS-Programm (ab Release 1.0) gebootet wurde und eine gültige ASSIGN.SYS Datei vorhanden ist.

Das geräteunabhängige GDOS (Graphics Device Operating System) enthält Grafikfunktionen und arbeitet mit geräteabhängigen Treibern für verschiedene Ausgabegeräte (Bildschirm, Drucker, Plotter, Metadatei usw.) zusammen.

Die ASCII-Datei ASSIGN.SYS enthält alle notwendigen Angaben über die verwendete Gerätekonfiguration. In dieser Datei müssen alle benutzten Gerätetreiber und Zeichensätze eingetragen werden, gegebenenfalls muß der Zugriffspfad für die Gerätetreiber angegeben werden, wenn sich diese nicht auf Laufwerk A: befinden. Folgende Syntax ist dabei zu beachten:

```
id    TREIBER.SYS;
;     Bemerkungen
      satz1.FNT
      satz2.FNT
      ...
      satzn.FNT

id    Enthält eine Zahl zwischen 1 und 32767. Damit wird
      der Typ des Gerätetreibers festgelegt, wobei folgende Zuordnung
      gilt:

      01 ...    Bildschirm
      11 ...    Plotter
      21 ...    Drucker
      31 ...    Metafile
      41 ...    Kamera
      51 ...    Grafik-Tablett
```

Befinden sich die Gerätetreiber auf Laufwerk C: im Ordner GEMSYS, dann sieht die zugehörige ASSIGN.SYS Datei z.B. so aus:

```
path = c:\gemsys\
01p screen.sys;   Bildschirmtreiber aus dem ROM, deshalb ein p hinter id
ATSS10.FNT
ATSS12.FNT
02p screen.sys;   Treiber für die niedrige Auflösung
ATSS10.FNT
ATSS12.FNT
03p screen.sys;   Treiber für die mittlere Auflösung
ATSS10CG.FNT
ATSS12CG.FNT
04p screen.sys;   Treiber für die hohe Auflösung
ATSS10.FNT
ATSS12.FNT
21 fx80.sys;      Druckertreiber für den FX-80 und Kompatible
ATSS10EP.FNT
ATSS12EP.FNT
31 META.SYS;      Metafile-Treiber
ATSS10MF.FNT
ATSS12MF.FNT
```

Dabei sind die Bildschirmtreiber SCREEN.SYS nur Dummy-Einträge, die die Syntaxkontrolle des GDOS so fordert. Die id's (2,3 und 4) haben nur die Bedeutung, jeder Auflösung entsprechende Fonts zuzuordnen. Deshalb ist der Aufruf zum Öffnen eines virtuellen Bildschirm-Workstation: V_OPNVWK(XBIOS(4)+2). Diesen Aufruf führt GFA-BASIC 3.0 intern durch.

GDOS?

GDOS? liefert TRUE, wenn GDOS (ab Release 1.0) geladen ist und FALSE, wenn GDOS nicht resident ist.

Beispiel:

```
IF NOT GDOS?
  ALERT 1,"GDOS nicht gefunden!",1,"Abbruch",r%
  END
ENDIF
```

V~H

Bei den folgenden, in GFA-BASIC implementierten, VDI-Kontrollfunktionen steht hd für das VDI-Handle und id für die Kennung eines Ausgabegerätes. In diesem Zusammenhang sind die folgenden Funktionen zu beachten:

```
V~H     Ergibt das GFA-BASIC intern benutzte VDI-Handle
        (z.B. PRINT V~H).
V~H=x   Setzt das intern benutzte (und das über CONTRL(6) erreichbare)
        VDI-Handle auf den Wert x.
V~H=-1  Setzt das intern benutzte (und das über CONTRL(6) erreichbare)
        VDI-Handle auf den Wert, den das intern benutzte V_OPNVWK
        ergeben hat.
```

V_OPNWK(id)
V_OPNWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSWK()

id: iexp
Die Zahlen 1 bzw. 2 zeigen die Default-Werte für die Einstellung der VDI-Parameter an, sie können auch geändert werden (vgl. WORK_OUT).

Die Funktion V_OPNWK (open workstation) liefert das Handle hd für die angegebene Gerätekennung id. Außerdem können weitere Informationen über das angeschlossene Gerät mit Hilfe von INTOUT() und PTSOUT() (vgl. VDISYS im Abschnitt Systemroutinen) erfragt werden.

Die Funktion V_CLSWK (close workstation) schließt die (mit V_OPNWK) geöffnete Arbeitsstation und schreibt alle Pufferinhalte dieser Arbeitstation. Außerdem wird V~H = -1 durchgeführt.

Beispiel: (vgl. bei V_CLRWK).

V_OPNVWK(id)
V_OPNVWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSVWK(id)

```
id: iexp
```

Die Funktion V_OPNVWK (open virtual workstation) öffnet einen virtuellen Bildschirmtreiber und liefert das Handle für die angegebene Gerätekennung id (vgl. V_OPNWK).

Die Funktion V_CLSVWK (close virtual workstation) schließt eine (mit V_OPNVWK) geöffnete virtuelle Workstation. Außerdem wird V~H=-1 durchgeführt.

V_CLRWK()
V_UPDWK()

Die Funktion V_CLRWK (clear workstation) löscht den Ausgabe-Puffer. Dabei werden z.B. der Bildschirm oder der Drucker-Puffer gelöscht.

Bei der Ausgabe von Grafik auf den Drucker werden alle Kommandos in einem Puffer gesammelt. Die Funktion V_UPDWK (update workstation) gibt diese gepufferten Grafikbefehle an das angeschlossene Gerät aus. Dagegen werden auf dem Bildschirm alle Grafikbefehle sofort ausgeführt.

Beispiel:

```
RESERVE 25600                    ! Genügend Speicherplatz reservieren
'
handle%=V_OPNWK(21)              ! Kennung für Ausgabegerät ermitteln
'
IF handle%=0
  ALERT 3,"Installationsfehler !",1,"Abbruch",r%
  RESERVE
  END
ENDIF
'
x_res%=INTOUT(0)                 ! x- und y-Auflösung des
y_res%=INTOUT(1)                 ! angeschlossenen Gerätes ermitteln
'
V~H=handle%                      ! setzt das interne VDI-Handle
'                                  auf Drucker-Kennung
~V_CLRWK()                       ! Puffer löschen
'
CLIP 0,0,x_res%,y_res%
```

```
BOX 0,0,x_res%,y_res%
LINE 0,0,x_res%,y_res%
LINE 0,y_res%,x_res%,0
'
~V_UPDWK()                          ! Grafikbefehle ausführen
~V_CLSWK()
'
RESERVE                             ! und den Speicher wieder zurückgeben
```

--> Gibt ein Rechteck, in dem die Diagonalen eingezeichnet sind auf einen angeschlossenen Drucker aus, wenn GDOS aktiv ist.

VST_LOAD_FONTS(x)
VST_UNLOAD_FONTS(x)

x: iexp

Die Funktion VST_LOAD_FONTS lädt die in ASSIGN.SYS angegebenen zusätzlichen Zeichensätze, wenn der zur Verfügung stehende Speicherplatz ausreicht und liefert die Anzahl der tatsächlich geladenen Schriften zurück. Stehen keine weiteren Fonts zur Verfügung, wird Null zurückgemeldet.

Der Parameter x sollte gleich Null sein, da Erweiterungen in anderen GEM-Versionen möglich sind. Wichtig ist vor allem, daß mit RESERVE ausreichend Speicherplatz für die zusätzlichen Zeichensätze reserviert wurde.

Die Funktion VST_UNLOAD_FONTS entfernt die zuvor mit VST_LOAD_FONTS geladenen Zeichensätze aus dem Speicher. Für den Parameter gilt das Gleiche wie bei VST_LOAD_FONTS beschrieben.

Beispiel:

```
RESERVE 25600
'
anz_fonts%=VST_LOAD_FONTS(0)      ! wie viele zusätzliche Zeichensätze?
face%=VQT_NAME(anz_fonts%,font$) ! Index und Name des geladenen Fonts
'
FOR i%=1 TO anz_fonts%
  DEFTEXT ,,,,face%
  TEXT 80,80,"Dies ist der "+font$+" Font."
  ~INP(2)
```

```
NEXT i%
'
~VST_UNLOAD_FONTS(0)                    ! Fonts entfernen
'
RESERVE
```

--> Gibt den Namen der geladenen Zeichensätze in diesen Fonts auf dem Bildschirm aus.

VQT_EXTENT(text$ [,x1,y1,x2,y2,x3,y3,x4,y4])

```
text$: sexp
x1,y1,x2,y2,x3,y3,x4,y4: ivar
```

Die Funktion VQT_EXTENT liefert die Ausmaße eines Rechtecks zurück, das den übergebenen String text$ einschließt. Wahlweise kann die Rückgabe in die Variablen x1,y1 bis x4,y4 erfolgen oder nur in PTSOUT(0) bis PTSOUT(7) erfolgen. Die Eckpunkte sind gegen den Uhrzeigersinn durchnumeriert.

Eckpunkt	Position
x1,y1	Linke, untere Ecke.
x2,y2	Rechte, untere Ecke.
x3,y3	Rechte, obere Ecke.
x4,y4	Linke, obere Ecke.

VQT_NAME(i,font_name$)

```
i: ivar
font_name$: svar
```

Die Funktion VQT_NAME liefert die Kennung des geladenen Zeichensatzes mit der Nummer i und trägt in der String-Variablen font_name$ den Namen dieses Fonts ein.

Beispiel:

```
RESERVE 2560
'
anz_fonts%=VST_LOAD_FONTS(0)
face%=VQT_NAME(anz_fonts%,font$) ! Index und Name eines der geladenen
Fonts
h%=12                            ! Texthöhe
s$="Beispieltext"
x0%=80                           ! Ausgabe-Koordinaten
y0%=80                           ! für s$
DEFTEXT 1,0,0,h%,face%
'
~VQT_EXTENT(s$,x1%,y1%,x2%,y2%,x3%,y3%,x4%,y4%)
'
GRAPHMODE 4
TEXT x0%,y0%,s$
PBOX x0%+x1%,x0%+y1%-h%-1,x0%+x3%,y0%+y3%-h%
'
~VST_UNLOAD_FONTS(0)             ! Fonts entfernen
'
RESERVE
```

--> Der String s$ wird auf dem Bildschirm am Punkt x0,y0 invertiert ausgegeben.

Fremdsprachen-Routinen-Aufruf

Die in diesem Abschnitt beschriebenen Kommandos dienen dem Aufruf von Unterroutinen, die in C oder Assembler geschrieben sind.

C:adr([x,y,...])

```
adr: avar (mindestens 32 Bit, besser adr%)
x,y: iexp
```

Die Funktion C ruft ein in C oder Assembler geschriebenes Unterprogramm auf, das an der Adresse adr steht. In den Klammern können die Parameter x,y,... übergeben werden. Die Parameterübergabe erfolgt wie in C. Die Parameter können als 32-Bit-Langwort oder 16-Bit-Wort übergeben werden, wobei die Übergabe als 16-Bit-Wert voreingestellt ist. Langworte können übergeben werden, indem man vor dem entsprechenden Parameter ein L: setzt. Beim Aufruf dieser Funktion werden auf dem Stack die Rücksprungadresse und dann die Parameter abgelegt. So führt z.B.:

```
VOID C:adr%(L:x,W:y,z)
```

auf dem Stack zu folgender Situation:

```
  (sp) --> Rücksprungadresse
 4(sp) --> x (4 Bytes)
 8(sp) --> y (2 Bytes)
10(sp) --> z (2 Bytes)
```

Der von der Funktion zurückgemeldete Wert ist der Inhalt des Registers d0 beim Rücksprung aus der Unterroutine (der mit RTS erfolgen muß).

Beispiel:

Das hier verwendete Assembler-Programm füllt einen Speicherbereich (z.B. ein Integer-Array) ab einer bestimmten Adresse mit den Zahlen (Langworte) von 0 bis n.

```
206F0004      move.l   4(sp),a0  ;Anfangsadresse
202F0008      move.l   8(sp),d0  ;Anzahl der Werte
7200          moveq.l  #0,d1     ;Zähler
6004          bra.s    ct_2      ;Einsprung in Schleife
20C1     ct_1:move.l   d1,(a0)+  ;Schleife, Wert schreiben
5281          addq.l   #1,d1     ;Zähler erhöhen
B081     ct_2:cmp.l    d1,d0     ;fertig ?
64F8          bcc.s    ct_1      ;nein, weiter -->
4E75          rts                ;Rücksprung ins GFA-BASIC
```

Das BASIC-Programm ist:

```
FOR i%=1 TO 11
  READ a%
  asm$=asm$+MKI$(a%)
NEXT i%
DATA $206F,$0004,$202F,$0008,$7200,$6004,$20C1,$5281,$B081,$64F8,$4E75
'
DIM x%(10000)
asm%=V:asm$
~C:asm%(L:V:x%(0),L:10000)
PRINT "z.B. x%(12): ";x%(12)
```

--> Füllt das Array x%() mit den Zahlen von 0 bis 10000. Bei dieser Anwendung entspricht die Ausführung der C:-Funktion den Befehlen:

```
FOR i%=1 TO n%
  x%(i%)=i%
NEXT i%
```

Eine andere Möglichkeit, ein Assembler-Programm in ein GFA-BASIC 3.0-Programm einzufügen, bietet der INLINE-Befehl. Zuerst ein File erzeugen, in dem das oben abgedruckte Assembler-Programm enthalten ist, z.B. durch (nach der oben abgedruckten READ-DATA-Schleife):

```
BSAVE "COUNT.INL",V:asm$,22
```

Dann das folgende Programm eingeben:

```
INLINE asm%,22
DIM x%(10000)
~C:asm%(L:V:x%(0),L:10000)
```

Dann in der Zeile, in der INLINE steht, die Help-Taste drücken, Load anwählen und COUNT.INL einladen. Nun kann man dieses Programm mit Save abspeichern.

MONITOR [x]

```
x: iexp
```

Dieser Befehl dient dem Aufruf von Assembler-Unterroutinen oder von Debug-Programmen oder anderen Hilfsprogrammen. Zu diesem Zweck muß der Illegal Instruktion Vektor (Adresse 16) auf die Adresse der Unterroutinen "gebogen" werden. Der MONITOR-Befehl erzeugt dann eine Illegal Instruction Exception, die zum Ansprung des Unterprogramms führt, das mit RTE (Return from Exception) enden muß. Der Parameter x ist ein Übergabewert, der ins Register d0 geschrieben wird.

Ein Anwendungsbeispiel ist in einem Programm wie dem bei C: beschriebenen einen Debugger zu verwenden. Dazu wird das GFA-BASIC vom Debugger aus geladen und gestartet. Dann, nach Eingabe des Programms, direkt hinter den INLINE-Befehl einfügen: MONITOR asm%. Nach Start des Programms wird sich der Debugger wegen "Illegal Instruction" melden. Nun kann man sich den Bereich, auf den d0 zeigt, diassemblieren lassen oder editieren. Anschließend kann man, eventuell nach Drücken von Shift-Alternate-Control, die Ausführung des BASIC-Programms mit dem GO-Befehl des Debuggers fortsetzen.

CALL adr([x,y,...])

```
adr: avar (mindestens 32 Bit, besser adr%)
x,y: iexp
```

Mit Hilfe des Befehls CALL können ebenfalls Assembler- oder C-Unterprogramme aufgerufen werden. Dabei ist 'adr' die Adresse, ab der das Assembler-Programm im Speicher liegt. Es besteht die Möglichkeit, eine Parameterliste zu übergeben. Nach dem Aufruf von CALL befindet sich die Rücksprungadresse auf dem Stack (die Assembler-Routine muß mit RTS enden). Dahinter folgt die Anzahl der übergebenen Parameter als 16-Bit-Wert und schließlich die Adresse, ab der die Parameter im Speicher liegen, als 32-Bit-Wert. Alle Parameter werden als Langworte (32 Bit) interpretiert.

Es besteht hier die Möglichkeit, auch Strings als Parameter zu übergeben. In diesem Fall ist der Übergabewert die Anfangsadresse des Strings.

Stackaufbau

```
 (sp) -->  Rücksprungadresse
4(sp) -->  Anzahl der Parameter (16 Bit)
6(sp) -->  Adresse des Parameterfeldes (32 Bit)
```

RCALL adr,reg%()

```
reg%(): Name 4-Byte-Integer-Arrays
adr: iexp
```

Der Befehl RCALL ermöglicht es, die Register vor dem Start der Assembler-Routine mit Werten vorzubelegen und die Registerinhalte nach Ablauf der Routine abzufragen. Diesem Zweck dient das Array reg%(), dessen Elemente vom 4-Byte-Integer-Typ sein müssen und das mindestens 16 Elemente haben muß. Vor dem Start der Assembler-Routine werden die Einträge dieses Arrays in die Register kopiert und nach Ablauf der Routine die Inhalte der Register in die entsprechenden Array-Elemente geschrieben. Dabei gilt folgende Zuordnung (bei OPTION BASE 0):

Datenregister	d0 bis d7	in reg%(0)	bis	reg%(7)
Adreßregister	a0 bis a6	in reg%(8)	bis	reg%(14)
User-Stack-Pointer	(a7)	in reg%(15)	(nur Rückgabe)	

Beispiel:

Das Assembler-Listing erwartet in a0 die Adresse des (logischen oder physikalischen) Bildschirmspeichers. Es invertiert dann den Bildschirm zwischen den y-Koordinaten, die in d0 und d1 übergeben werden. Auf dem Farbbildschirm werden die Koordinaten verdoppelt.

```
 sub    d0,d1        ;Anzahl der Linien
 mulu   #20,d1       ;Schleifenabbruchkriterium
 subq   #1,d1        ; "
 mulu   #80,d0       ;Anzahl zu überspringender Bytes
 add.l  d0,a0        ;Adresse, ab der invertiert wird
schleife             ;Schleifenanfang
 not.l (a0)+         ;invertiere 4 Byte
 dbra d1,schleife    ;dekremetiere und verzweige
 rts                 ;zurück ins GFA-BASIC
```

Das aufrufende GFA-BASIC-Programm lautet:

```
DO
  READ a%
  EXIT IF a%=-1
  A$=A$+MKI$(a%)
LOOP
DATA 37440,49916,20,21313
DATA 49404,80,53696
DATA 18072,20937,65532,20085,-1
'
DIM r%(16)
xb2%=XBIOS(2)
HIDEM
FOR j%=1 TO 50
  FOR i%=0 TO 190 STEP 10
    r%(0)=i%
    r%(1)=399-i%
    r%(8)=xb2%
    RCALL V:a$,r%()
  NEXT i%
NEXT j%
```

--> Das Programm produziert eine grafische Spielerei. Natürlich ist es auch hier, wie bei C:, möglich, INLINE zu verwenden.

EXEC mod,nam,cmdl,envs
EXEC(mod,nam,cmdl,envs)

```
mod: iexp
nam,cmdl,envs: sexp
```

Die Anweisung EXEC kann als Befehl, aber auch als Funktion verwendet werden. Sie dient dem Laden und Starten von Programmen von Diskette, die nach ihrer Beendigung wieder zum aufrufenden Programm zurückkehren. Vor dem Aufruf von EXEC muß dem aufzurufenden Programm Speicherplatz zur Verfügung gestellt werden (siehe Beispiel). Der Parameter mod gibt den Modus des Aufrufs an. Dabei bedeutet:

```
0 --> Programm laden und starten.
3 --> Programm nur laden.
```

Im Ausdruck nam steht der Name des zu ladenden (und zu startenden) Programms. Das Format der Namensangabe entspricht den Regeln des hierarchischen File-Systems, die im Kapitel über allgemeine Ein- und Ausgabe beschrieben sind.

Der Ausdruck cmdl enthält die Kommadozeile, die in der Basepage des aufgerufenen Programms eingetragen wird. Das erste Zeichen der Kommandozeile enthält die Länge derselben (maximal 127). In der Regel wird dieses Byte ignoriert, es sei denn, es wäre Null. Deshalb reicht es normalerweise aus, ein Dummy-Zeichen (*) einzufügen.

In envs steht die Environment. Dies ist ein durch CHR$(0) unterteilter String. Für ein C-Programm also eine Reihe von Strings, deren letzter durch ein doppeltes Nullbyte (macht das GFA-BASIC selber) gekennzeichnet wird. Dieses Environment wird von vielen Shell-Programmen benutzt, um Variablen und ihre Namen zu speichern (meistens SET-Befehl). Auch viele Compiler benutzen dies, um Zugriffspfade oder ähnliches festzulegen.

Ruft man EXEC als Funktion auf, so erhält man den Rückgabewert des Programms oder, für mod=3, die Adresse der Basepage des aufgerufenen

Programms. EXEC 3 ist nur für speziell zu diesem Zweck geschriebene Programme (Overlays) sinnvoll anwendbar, oder für Debugger.

Beispiel:

```
FILESELECT "\*.PRG","",f$
RESERVE 100
SHOWM
a%=EXEC(0,f$,"","")
RESERVE
PRINT "Zurück im GFA-BASIC, Rückgabewert",a%
```

--> Gestattet das Aufrufen eines Programms (sofern es nicht zuviel Speicherplatz benötigt) und kehrt nach der Beendigung des aufgerufenen Programms wieder in den Interpreter zurück. Der Rückgabewert, der von der EXEC-Funktion geliefert wird, kann auch von GFA-BASIC-Programmen über QUIT n oder SYSTEM n festgelegt werden.

11 - AES-Bibliotheken

Das folgende Kapitel soll einen Überblick über die AES-Bibliotheken (AES = Application Environment Services) geben. Eine ausführliche Beschreibung dieser Routinen würde allerdings den Rahmen eines Handbuchs sprengen, deshalb sei an dieser Stelle auf die umfangreiche Literatur zum Thema GEM verwiesen. In diesem Kapitel werden daher nur Informationen in komprimierter Form gegeben. Die einzelnen Routinen sind nach folgendem Schema aufgeführt worden:

In der ersten Zeile steht der Name der entsprechenden Bibliotheks-Routinen, danach folgt eine kurze Beschreibung der Funktion. Dahinter wird der komplette Funktionsaufruf in GFA-BASIC-3.0-Notation aufgeführt und die Bedeutung aller verwendeteten Variablen erklärt.

Das Kapitel schließt mit einigen längeren Beispielprogrammen.

Bevor mit der Beschreibung der Befehle aus den 11 AES-Bibliotheken begonnen wird, müssen die wichtigsten Datenstrukturen bekannt sein, mit denen die AES arbeiten. Dies sind die OBJEKT-, TEDINFO-, ICOBLK, BITBLK-, USERBLK- und PARMBLK-Strukturen.

Zu allererst eine Aufstellung der AES-Befehle, die in ähnlicher Form auch im GFA-BASIC 2.0 enthalten sind.

GCONTRL
ADDRIN
ADDROUT
GINTIN
GINTOUT
GB

Diese Funktionen ermitteln die Adressen der AES-Parameterblöcke. Das sind im einzelnen:

```
GCONTRL --> Adresse des AES-Control-Arrays.
ADDRIN  --> Adresse des AES-Adreß-Input-Arrays.
ADDROUT --> Adresse des AES-Adreß-Output-Arrays.
GINTIN  --> Adresse des AES-Integer-Input-Arrays.
GINTOUT --> Adresse des AES-Integer-Output-Arrays.
GB      --> Adresse der AES-Parameter-Arrays.
```

Mit Index in Klammern greifen diese Funktionen direkt auf die entsprechenden Felder zu. GCONTRL(3) entspricht DPEEK(GCONTRL+6) bzw. dem entsprechendem DPOKE.

Dabei sind GCONTRL, GINTIN und GINTOUT wortweise organisiert, ADDRIN und ADDROUT in Langworten.

Also entspricht ADDRIN(2)=x dann LPOKE ADDRIN+2*4,x.

Das GB-Feld kann nicht mit Indizes verwendet werden, da es nur aus Kompatiblitätsgründen zum ST-BASIC vorhanden ist - GINTIN usw. sind ja vorhanden. Das zweite Langwort ({GB+4}) ist die Adresse des GEM-internen global-Feldes.

GEMSYS nr

nr: iexp

Mit Hilfe des Befehls GEMSYS kann eine AES-Routine mit Hilfe ihrer Funktionsnummer aufgerufen werden. Die für diese Routine notwendigen Parameter müssen in den AES-Parameterblöcken abgelegt werden.

Beispiel:

```
REPEAT
  GINTIN(0)=60
  GINTIN(1)=30
  GINTIN(2)=10
  GINTIN(3)=10
  GINTIN(4)=200
  GINTIN(5)=200
  GEMSYS 72
UNTIL MOUSEK
```

--> Zeichnet ein sich bewegendes Rechteck auf den Bildschirm (72 ist die Nummer der Funktion GRAF_MOVEBOX).

Objekt-Struktur

Offset	Inhalt	Typ	Bedeutung
00	ob_next	word	Zeiger auf das folgende Objekt.
02	ob_head	word	Zeiger auf das erste Kind.
04	ob_tail	word	Zeiger auf das letzte Kind.
06	ob_type	word	Typ des Objekts.
08	ob_flags	word	Objektinformationen (siehe unten).
10	ob_state	word	Status des Objekts (siehe unten).
12	ob_spec	long	Zeiger auf weitere Informationen (s.u.).
16	ob_x	word	X-Position des Objekts.
18	ob_y	word	Y-Position des Objekts.
20	ob_w	word	Breite des Objekts.
22	ob_h	word	Höhe des Objekts.

Der Speicherplatzbedarf eines Objekts umfaßt also diese 24 Byte plus eventuell vorhandene weitere Beschreibungsstrukturen, z.B. TEDINFO- oder BITBLK-Strukturen.

OB_NEXT Zeiger auf das folgende Objekt in der gleichen Ebene oder auf das Mutterobjekt, falls es sich um das letzte Element in der Ebene handelt.

OB_HEAD Zeiger auf das erste Kind des entsprechenden Objekts.

OB_TAIL Zeiger auf das letzte Kind des entsprechenden Objekts.

Wenn keine tieferen Ebenen mehr vorliegen, haben die Zeiger den Wert -1 auch NIL (not in list) genannt.

Folgende OB_TYPE-Werte sind möglich. Je nach Wert von OB_TYPE ist OB_SPEC die Adresse verschiedener Informationsstrukturen.

OB_TYPE		OB_SPEC
G_TEXT	21	Adresse einer TEDINFO-Struktur.
G_BOXTEXT	22	Adresse einer TEDINFO-Struktur.
G_IMAGE	23	Adresse einer BITBLK-Struktur.
G_USERDEF	24	Adresse einer USERBLK-Struktur.
G_IBOX	25	BOXINFO, siehe unten.
G_BUTTON	26	Adresse des Strings, der den Text enthält.
G_BOXCHAR	27	BOXINFO, siehe unten.
G_STRING	28	Adresse des entsprechenden Strings.
G_FTEXT	29	Adresse einer TEDINFO-Struktur.
G_FBOXTEXT	30	Adresse einer TEDINFO-Struktur.
G_ICON	31	Adresse einer ICONBLK-Struktur.
G_TITLE	32	Adresse des entsprechenden Strings.

Für G_BOX, G_IBOX und G_BOXCHAR enthält OB_SPEC Informationen, die Zeichen, Rahmen und Objektfarbe des entsprechenden Objekts betreffen. Dabei sind die obersten 8 Bits nur bei G_BOXCHAR belegt und enthalten dort den ASCII-Code des in der Box erscheinenden Zeichens.

Sie enthalten folgende Werte für den Rahmen:

0 =	Kein Rahmen.
1 bis 128 =	Der Rand liegt 1 bis 128 Pixel innerhalb des entsprechenden Objekts.
-1 bis -128 =	Der Rand liegt 1 bis 128 Pixel außerhalb des entsprechenden Objekts.

Bit-Belegung für die Objektfarbe: 1111 2222 3444 5555

1	Rahmen	(0 bis 15)
2	Text	(0 bis 15)
3	Text	(0 = durchsichtig, 1 = deckend)
4	Füllmuster	(0 bis 7)
5	Farbe des Objektinneren	

OB_FLAGS	Hex	Bit-Nr.
SELECTABLE	&H0001	0
DEFAULT	&H0002	1
EXIT	&H0004	2
EDITABLE	&H0008	3
RBUTTON	&H0010	4
LASTOB	&H0020	5
TOUCHEXIT	&H0040	6
HIDETREE	&H0080	7
INDIRECT	&H0100	8

OB_STATE	Hex	Bit-Nr.
SELECTED	&H0001	0
CROSSED	&H0002	1
CHECKED	&H0004	2
DISABLED	&H0008	3
OUTLINED	&H0010	4
SHADOWED	&H0020	5

Die im vorangegangenen Abschnitt erläuterten Strukturen werden in GFA-BASIC 3.0 mit folgender Syntax angesprochen (lesen und schreiben):

OB_NEXT(tree%,obj&)
OB_HEAD(tree%,obj&)
OB_TAIL(tree%,obj&)

OB_TYPE(tree%,obj&)
OB_FLAGS(tree%,obj&)
OB_STATE(tree%,obj&)
OB_SPEC(tree%,obj&)

OB_X(tree%,obj&)
OB_Y(tree%,obj&)
OB_W(tree%,obj&)
OB_H(tree%,obj&)

Dabei ist tree% die Adresse des Objektbaumes und obj& die Nummer des Objekts, über das Informationen gewünscht werden.

Außerdem kann die Adresse des Objekts mit

OB_ADR(`tree%,obj&`)

bestimmt werden.

Textinformations-Struktur (TEDINFO)

Offset	Inhalt	Typ	Bedeutung
00	te_ptext	long	Zeiger auf Text.
04	te_ptmplt	long	Zeiger auf Textmaske.
08	te_pvalid	long	Zeiger auf String für Gültigkeit der Eingabezeichen.
12	te_font	word	Zeichensatz.
14	te_resvd	word	Reserviert.
16	te_just	word	Textausrichtung.
18	te_color	word	Farbe der umgebenden Box.
20	te_resvd2	word	Reserviert.
22	te_thickness	word	Rahmenstärke.
24	te_txtlen	word	Textlänge.
26	te_tmplen	word	Textmaskenlänge.

Icon-Struktur (ICONBLK)

Offset	Inhalt	Typ	Bedeutung
00	ib_pmask	long	Zeiger auf Icon-Maske.
04	ib_pdata	long	Zeiger auf Icon-Daten.
08	ib_ptext	long	Zeiger auf Icon-Text.
12	ib_char	word	Zeichen innerhalb des Icons.
14	ib_xchar	word	x-Position des Zeichens.
16	ib_ychar	word	y-Position des Zeichens.
18	ib_xicon	word	x-Position des Icons.
20	ib_yicon	word	y-Position des Icons.
22	ib_wicon	word	Breite des Icons.
24	ib_hicon	word	Höhe des Icons.
26	ib_xtext	word	x-Position des Textes.
28	ib_ytext	word	y-Position des Textes.
30	ib_wtext	word	Textbreite in Pixel.
32	ib_htext	word	Texthöhe in Pixel.
34	ib_resvd	word	Reserviert.

Bit-Image-Struktur (BITBLK)

Offset	Inhalt	Typ	Bedeutung
00	bi_pdata	long	Zeiger auf Image-Daten.
04	bi_wb	word	Breite des Images in Bytes.
06	bi_hl	word	Höhe des Images in Pixeln.
08	bi_x	word	x-Position des Images.
10	bi_y	word	y-Position des Images.
12	bi_color	word	Farbe des Images.

Applikations-Block-Struktur (USERBLK)

Offset	Inhalt	Typ	Bedeutung
00	ub_code	long	Zeiger auf eine selbstdefinierte Routine zur Darstellung des Objekts.
04	up_parm	long	Zeiger auf eine PARMBLK-Struktur.

Dabei muß es sich um Assembler-Routinen handeln.

Parameter-Block-Struktur (PARMBLK)

Offset	Inhalt	Typ	Bedeutung
00	pb_tree	long	Zeiger auf Objektbaum.
04	pb_obj	word	Objektnummer.
06	pr_prevstate	word	Vorheriger Status.
08	pr_currstate	word	Aktueller Status.
10	pb_x	word	x-Position des Objekts.
12	pb_y	word	y-Position des Objekts.
14	pb_w	word	Breite.
16	pb_h	word	Höhe.
18	pb_xc	word	x-Position des Clipping-Rechtecks.
20	pb_yc	word	y-Position des Clipping-Rechtecks.
22	pb_wc	word	Breite des Clipping-Rechtecks.
24	pb_hc	word	Höhe des Clipping-Rechtecks.
28	pb_parm	long	Parameter aus der USERBLK-Struktur.

Bei den AES-Funktionen ist zu beachten, daß manche Funktionen nicht nur einen Funktionswert liefern, sondern auch Rückgabewerte in dafür bereitgestellte Variablen zurückgeben. Andere Funktionen liefern reservierte Werte und können über VOID bzw. dessen Verkürzung, die Tilde "~" aufgerufen werden. Bei der Angabe von Adressen müssen mindestens 4-Byte lange Variablentypen verwendet werden, und Koordinatenangaben müssen in mindestens 2-Byte langen Variablentypen erfolgen.

Applikations-Bibliothek (application library)

Steuert die Zugriffe auf andere AES-Bibliotheken. Die Funktionen APPL_INIT und APPL_EXIT werden von GFA-BASIC 3.0 automatisch beim Starten des Programms bzw. beim Verlassen aufgerufen.

APPL_INIT()

Das aktuelle Programm wird als GEM-Applikation angemeldet. Die Funktion APPL_INIT liefert das handle des aktuellen GFA-BASIC-Programms. Diese Identifikationsnummer ist zum Beispiel wichtig für die Installation zusätzlicher Fonts.

rueck Enthält die Identifikationsnummer (ap_id) der Applikation (des Programms) für das GEM. APPL_INIT wird für APPL_READ, APPL_WRITE und MENU_REGISTER benutzt. Diese Funktion ist in GFA-BASIC eine Dummy-Funktion, die keinen Betriebssystemaufruf durchführt, da die APPL_INIT bereits beim Start des GFA-BASIC-Interpreters ausgeführt wird. Die dabei erhaltene ap_id wird zurückgegeben.

APPL_READ(id,len,adr_buffer)

```
id,len,addr_buffer: iexp
```

Mit diesem Befehl können Bytes aus einem Ereignispuffer (Message-Buffer) gelesen werden.

rueck	0 = Fehler aufgetreten.
id	Nummer der Applikation, aus deren Puffer gelesen wird.
len	Anzahl zu lesender Bytes.
adr_buffer	Adresse, ab der die gelesenen Bytes abgelegt werden.

APPL_WRITE(id,len,adr_buffer)

```
id,len,addr_buffer: iexp
```

Damit werden Bytes in einen Ereignispuffer geschrieben.

```
rueck        0 = Fehler aufgetreten.
id           Nummer der Applikation, in deren Puffer geschrieben wird.
len          Anzahl zu schreibender Bytes.
adr_buffer   Adresse, an die die Bytes geschrieben werden.
```

APPL_FIND(fname$)

```
fname$: sexp
```

Ermittelt die Identifikationsnummer einer Applikation, deren Dateiname bekannt ist, z.B. zum Informationsaustausch mit anderen laufenden Programmen.

```
rueck   Identifikation des gesuchten Programms,
        -1 = Fehler aufgetreten, Programm nicht gefunden.
fname   Enthält den 8 Zeichen langen Dateinamen (ohne Extension)
        der zu suchenden Applikation. Der Dateiname muß unbedingt
        8 Zeichen lang und in Großbuchstaben geschrieben sein und
        gegebenenfalls mit Leerzeichen aufgefüllt werden.
```

Beispiel:

```
PRINT APPL_FIND("CONTROL ")
```

--> Liefert 2 (o.ä.), wenn das Control-Accessory geladen ist oder -1, wenn nicht. Außer Accessorys kann man das eigene Programm "GFABASIC" (ap_id = 0) und den Screen-Manager "SCRENMGR" (ap_id = 1), der z.B. für das Menü-Handling zuständig ist, suchen.

APPL_TPLAY(mem,num,scale)
APPL_TRECORD(mem,num)

```
mem,num,scale: iexp
```

Sollen, nach Art eines Tonbandes, Tape play und record, eine Aufzeichnung von Benutzeraktivitäten (Maus- Tastenbetätigungen und den Abstand dazwischen) abspielen und erstellen können. Diese Funktionen arbeiten nicht so, wie in der GEM-Dokumentation angegeben.

Mit neueren ROM-Versionen arbeiten diese Funktionen zwar annähernd wie angegeben (statt 6 Byte pro Ereignis aufzuzeichenen werden 8 Byte verbraucht und auch der scale-Faktor wirkt anders), da aber dieses Verhalten alles andere als zuverlässig erscheint, wird von der Verwendung dieser Funktionen dringendst abgeraten. Daher entfällt auch jede weitere Beschreibung.

APPL_EXIT()

APPL_EXIT meldet ein Programm bei den AES ab. Dabei wird die Identifikationsnummer des entsprechenden Programms freigegeben und steht somit für andere Programme wieder zur Verfügung.

APPL_EXIT existiert in GFA-BASIC nur als Dummy-Funktion, da ein QUIT/SYSTEM-Befehl dies automatisch durchführt.

```
rueck     0 = Fehler aufgetreten
```

Ereignis-Bibliothek (event library)

Reagiert auf Eingaben über Maus, Tastatur oder zeitabhängige Ereignisse.

EVNT_KEYBD()

Wartet auf ein Tastatur-Ereignis und gibt den entsprechenden Tastatur-Code zurück.

rueck Tasten-Code der gedrückten Taste (low byte) und Scan-Code (high byte).

Beispiel:

```
DO
  PRINT HEX$(EVNT_KEYBD(),4)
LOOP
```

--> Gibt diesen Tasten-Code in hex aus.

EVNT_BUTTON(clicks,mask,state [,mx,my,button,k_state])

```
clicks,mask,state: iexp
mx,my,button,k_state: ivar
```

Wartet auf ein festzulegendes Maustasten-Ereignis.

rueck	Anzahl der getätigten Maustastendrücke.
clicks	Anzahl der notwendigen Maustastendrücke.
mask	Maske für die gewünschte Maustaste: Bit 0 = links, Bit 1 = rechts
state	Gewünschter Status, um das Ereignis auszulösen. Bit-Belegung wie bei mask.
mx	x-Koordinate bei Ereignisauslösung durch Maus-Cursor.
my	y-Koordinate bei Ereignisauslösung durch Maus-Cursor.
button	Status der Maustasten, wie bei state.
k_state	Zustand der Tastaturumschalttasten bei Auslösung des Ereignisses: Bit 0 = Rechte Shift-Taste Bit 1 = Linke Shift-Taste Bit 2 = Control-Taste Bit 3 = Alternate-Taste

Die Parameter mx,my,button und k_state sind optional, diese Werte können auch über GINTOUT(1) bis GINTOUT(4) abfragt werden.

Beispiel:

```
DO
  SELECT EVNT_BUTTON(2,1,1,mx%,my%,bu%,kb%)
  CASE1
    PLOT mx%,my%
  CASE2
    PBOX mx%-10,my%-10,mx%+10,my%+10
  ENDSELECT
LOOP UNTIL BTST(kb%,2)
```

--> Wartet auf Mausklicks mit der linken Maustaste. Bei Einfachklick wird ein Punkt gesetzt, bei Doppelklick ein Quadrat. Beendet wird dieses Beispiel, wenn zusätzlich zum Klick auch die Control-Taste gedrückt wird.

EVNT_MOUSE(flags,mx,my,mw,mh,mcur_x,mcur_y,button,k_state)

```
flags,mx,my,mw,mh: iexp
mcur_x,mcur_y,button,k_state: ivar
```

Wartet, bis der Mauszeiger einen rechteckigen Bildschirmteil betritt oder verläßt.

rueck	Reserviert, immer 1.
flags	Meldet das Betreten (0) oder Verlassen (1) des rechteckigen Bildschirmteils.
mx	Linke x-Koordinate.
my	Obere y-Koordinate.
mw	Breite.
mh	Höhe des rechteckigen Bildschirmteils.
mcur_x	x-Koordinate.
mcur_y	y-Koordinate des Maus-Cursors.
button	Maustastenstatus bei Eintritt des Ereignisses: Bit 0 = Rechte Maustaste. Bit 1 = Linke Maustaste.
k_state	Tastaturzustand beim Auftreten des Ereignisses: Bit 0 = Rechte Shift-Taste. Bit 1 = Linke Shift-Taste. Bit 2 = Control-Taste Bit 3 = Alternate-Taste

Beispiel:

```
DO
  EVNT_MOUSE(0,100,100,200,90,mx%,my%,bu%,kb%)
  IF bu%
    PBOX mx%-10,my%-10,mx%+10,my%+10
  ELSE
    PLOT mx%,my%
  ENDIF
LOOP UNTIL BTST(kb%,2)
```

--> Es wird gewartet, bis sich der Maus-Cursor in einem Rechteck befindet. Dann wird bei gedrückter Maustaste ein Rechteck gezeichnet, sonst ein Punkt. Ende durch Drücken von Control, wenn Mauszeiger innerhalb eines Rechtecks ist.

Achtung: Während der Ausführung von GEM-Routinen kann zwar Shift-Control-Alternate gedrückt werden, eine Auswirkung hat dies aber erst nach Rückkehr ins BASIC.

So legt zum Beispiel ~EVNT_TIMER(3600000) oder DELAY 3600 eine Stunde Pause ein, bis der Rechner aus dem GEM wiederkommt.

EVNT_MESAG(adr_buffer)

```
adr_buffer: iexp
```

Wartet auf das Eintreffen einer Meldung (message) im Ereignispuffer.

```
rueck        Reserviert, immer 1.
adr_buffer   Adresse des 16 Byte langen Puffers (message buffer) zur
             Aufnahme der Meldung (vgl. MENU(1) bis MENU(8)).
             Bei Angabe von 0 für adr_buffer wird selbständig der
             GFA-BASIC-eigene Message-Puffer verwendet, also MENU(1)
             bis MENU(8).
```

EVNT_TIMER(count)

```
count: iexp
```

Wartet eine in Millisekunden anzugebende Zeitspanne (vgl. DELAY).

```
rueck     Reserviert, immer 1.
count     Anzahl von Millisekunden.
```

EVNT_MULTI(flags,clicks,mask,state,m1_flags,m1_x,m1_y,m1_w,m1_h,
m2_flags,m2_x,m2_y,m2_w,m2_h,adr_buffer,count
[,mcur_x,mcur_y,button,k_state,key,anz_clicks])

```
flags,clicks,mask,state,m1_flags,m1_x,m1_y,m1_w,m1_h: iexp
m2_x,m2_y,m2_w,m2_h,adr_buffer,count: iexp
mcur_x,mcur_y,button,k_state,key,anz_clicks: ivar
```

Wartet auf das Eintreten verschiedener Ereignisse.

rueck — Enthält das Ereignis, welches tatsächlich eingetreten ist, Bit-Belegung wie bei ev_mflags.

flags — Legt fest, auf welches Ereignis die Applikation warten soll. Dabei entspricht:

Bit 0	Tastatur	MU_KEYBD
Bit 1	Mausknopf	MU_BUTTON
Bit 2	Erstes Mausereignis	MU_M1
Bit 3	Zweites Mausereignis	MU_M2
Bit 4	Mitteilungs-Ereignis	MU_MESAG
Bit 5	Timer	MU_TIMER

anz_clicks — Anzahl der getätigten Maustastendrücke.

Die folgenden Parameter wurden bereits bei EVNT_MOUSE, EVNT_KEYBD, EVNT_BUTTON und EVNT_MESAG erläutert. Jedoch ist zu beachten, daß auf zwei verschiedene Mausereignisse (m1 und m2) gewartet werden kann. Bei ON MENU, das intern diese Routine verwendet, werden die Parameter über ON MENU xxx GOSUB eingestellt bzw. count direkt im Befehl angegeben.

MENU(1) bis MENU(8)	Message-Puffer
MENU(9)	Rückgabewert
MENU(10) = mcur_x	Maus-Cursor-Position
MENU(11) = mcur_y	Maus-Cursor-Position
MENU(12) = button	Maus-Button
MENU(13) = k_state	BIOS(11,-1)
MENU(14) = key	Scan- und ASCII-Codes
MENU(15) = anz_clicks	Anzahl der Maus-Klicks

EVNT_DCLICK(new,get_set)

```
new,get_set: iexp
```

Legt die Geschwindigkeit für Doppelklicks der Maus fest.

```
rueck    Alte Geschwindigkeit.
new      Neue Geschwindigkeit (0 bis 4).
get_set  Art der Einstellung:
          0 = Aktuellen Wert ermitteln, ohne neu zu beachten.
          1 = Neuen Wert für new festlegen.
```

Menü-Bibliothek (menu library)

Übernimmt das Zeichnen einer Menüleiste und deren Verwaltung.

MENU_BAR(tree,flag)

```
tree,flag: iexp
```

Schaltet eine Menüleiste (aus einem Resource-File) ein oder aus.

Vergleiche: MENU x$() und MENU KILL.

```
rueck   0 = Fehler aufgetreten.
tree    Adresse des Menüobjektbaumes.
flag    1 = Menüleiste einschalten.
        0 = Menüleiste ausschalten (immer zum Programmende).
```

MENU_ICHECK(tree,item,flag)

```
tree,item,flag: iexp
```

Setzt oder löscht einen Haken vor einem Menüeintrag (dazu sollten vor jedem Menüeintrag mindestens zwei Leerzeichen stehen).

Vergleiche: MENU x,0 und MENU x,1.

```
rueck   0 = Fehler aufgetreten.
tree    Adresse des Menüobjektbaumes.
item    Objektnummer des entsprechenden Menüeintrags.
flag    0 = Haken setzen.
        1 = Haken löschen.
```

MENU_IENABLE(tree,item,flag)

```
tree,item,flag: iexp
```

Bewirkt das Ein- oder Ausschalten von Menüeinträgen. Der entsprechende Menüeintrag wird dann in hellgrauer Schrift dargestellt.

Vergleiche: MENU x,2 und MENU x,3.

```
rueck   0 = Fehler aufgetreten.
tree    Adresse des Menüobjektbaumes.
item    Objektnummer des entsprechenden Menüeintrags.
flag    0 = Ausschalten.
        1 = Einschalten.
```

MENU_TNORMAL(tree,title,flag)

```
tree,title,flag: iexp
```

Schaltet Menütitel auf invertierte bzw. normale Darstellung.

Vergleiche: MENU OFF.

rueck	0 = Fehler aufgetreten.
tree	Adresse des Menüobjektbaumes.
title	Objektnummer des entsprechenden Menütitels.
flag	0 = Invertierte Darstellung.
	1 = Normale Darstellung.

MENU_TEXT(tree,item,new_text$)

```
tree,item: iexp
new_text$: sexp
```

Ändert den Text eines Menüeintrags. Diese Funktion erlaubt die Anpassung von Menüeinträgen an den aktuellen Programmzustand.

rueck	0, wenn Fehler aufgetreten.
tree	Adresse des Menüobjektbaumes.
item	Nummer des zu ändernden Objekteintrags.
new_text$	String, der den neuen Menüeintrag enthält (darf die Länge des alten Eintrags nicht überschreiten).

MENU_REGISTER(ap_id,m_text$)

```
ap_id: iexp
m_text$: sexp
```

Trägt die Namen eines Accessories unter dem ersten Menütitel ein. Dabei können maximal 6 Accessory-Menüeinträge eingerichtet werden.

Diese Funktion kann jedoch nur sinnvoll von einem Accessory angewandt werden (nur im compilierten Programm).

```
rueck     Kennung für den Menüeintrag des entsprechenden
          Accessories:
           0 bis 5 für den ersten bis sechsten Accessory-Eintrag.
          -1, wenn kein weiterer Eintrag möglich ist.
ap_id     Identifikationsnummer des entsprechenden Accessories.
m_text$   Adresse des entsprechenden Menüeintrags.
```

Beispiel: Entfällt, da nur in Accessories möglich.

Objekt-Bibliothek (object library)

Enthält Routinen zur Definition, Darstellung und Veränderung von Objekten.

OBJC_ADD(tree,parent,child)

```
tree,parent,child: iexp
```

Fügt ein Objekt innerhalb eines bestimmten Objektbaumes hinzu und stellt die Verknüpfung zwischen den bestehenden Objekten und dem neuen Objekt her.

```
rueck      0 = Fehler aufgetreten.
tree_adr   Adresse des entsprechende Objektbaumes.
parent     Objektnummer des "Elternobjekts", zu dem das neue Objekt
           hinzugefügt wird.
child      Objektnummer des "Kindobjekts", das dem "Elternobjekt"
           angegliedert wird.
```

OBJC_DELETE(tree,del_obj)

```
tree,del_obj: iexp
```

Verändert die Next-, Head- und Tail-Zeiger eines Objekts innerhalb des entsprechenden Objektbaumes, so daß es nicht mehr angewählt werden kann, bis diese Zeiger wieder restauriert werden.

rueck	0 = Fehler aufgetreten.
tree	Adresse des entsprechenden Objektbaumes.
del_obj	Objektnummer des zu löschenden Objekts.

OBJC_DRAW(tree,start_obj,depth,cx,cy,cw,ch)

```
tree,star_obj,depth,cx,cy,cw,ch: iexp
```

Diese Funktion stellt ganze Objekte oder deren Teile auf dem Bildschirm dar. Wahlweise können die Koordinaten eines rechteckigen Bildschirmausschnittes angegeben werden, auf den die Darstellung beschränkt wird.

rueck	0 = Fehler aufgetreten.
tree	Adresse des entsprechenden Objektbaumes.
start_obj	Objektnummer des ersten darzustellenden Objekts. 0 = Nur erstes Objekt zeichnen.
depth	Anzahl der darzustellenden Objektebenen.
cx	Linke x-Koordinate.
cy	Obere y-Koordinate.
cw	Breite und
ch	Höhe des Begrenzungsrechtecks.

OBJC_FIND(tree,start_obj,depth,fx,fy)

```
tree,star_obj,depth: iexp
fx,fy: ivar
```

Diese Funktion ermöglicht die Angabe von Maus-Cursor-Koordinaten und ermittelt die Objektnummer des Objektes, daß sich an der angegebenen Bildschirmposition befindet.

```
rueck       Nummer des gefundenen Objekts.
             -1 = keine Objektnummer gefunden.
tree        Adresse des zu untersuchenden Objektbaumes.
start_obj   Nummer des ersten zu untersuchenden Objekts.
depth       Anzahl der zu untersuchenden Objektebenen.
             0 = nur das "Mutterobjekt".
fx          x-Koordinate und
fy          y-Koordinate der anzugebenden Bildschirmposition.
```

OBJC_OFFSET(tree,obj,x_abs,y_abs)

```
tree,obj: iexp
x_abs,y_abs: ivar
```

Berechnet die absoluten Bildschirmkoordinaten eines anzugebenden Objekts.

rueck	0 = Fehler aufgetreten.
tree	Adresse des entsprechenden Objektbaumes.
obj	Nummer des anzugebenden Objekts.
x_abs	Die berechneten absoluten x- und
y_abs	y-Koordinaten.

OBJC_ORDER(tree,obj,new_pos)

```
tree,obj,new_pos: iexp
```

Ändert die Reihenfolge eines anzugebenden Objekts innerhalb der Objektebenen.

```
rueck      0 = Fehler
tree       Adresse des entsprechenden Objektbaumes.
obj        Nummer des zu ändernden Objekts.
new_pos    Nummer der neuen Objektebene:
            -1 = Eine Ebene höher
             0 = Unterste Ebene
             1 = Unterste Ebene + 1
             2 = Unterste Ebene + 2
                 usw.
```

OBJC_EDIT(tree,obj,char,old_pos,flag,new_pos)

```
tree,obj,char,old_pos,flag: iexp
new_pos: ivar
```

Ermöglicht die Eingabe und das Editieren von Texten in die Objekttypen G_TEXT und G_BOXTEXT.

```
rueck     0 = Fehler aufgetreten.
tree      Adresse des entsprechenden Objektbaumes.
obj       Objektnummer des entsprechenden Objekts.
char      Eingegebenes Zeichen (incl. Scan-Code).
old_pos   Position des Zeichens im Eingabe-String.
flag      Funktionsauswahl:
          0   ED_START = Reserviert - Dummy-Aufruf.
          1   ED_INIT  = Formatierten String berechnen und Cursor
                         einschalten.
          2   ED_CHAR  = Betreffendes Zeichen verarbeiten und den
                         String erneut anzeigen.
          3   ED_END   = Cursor ausschalten.
new_pos   Neue Position des Zeichens im Eingabe-String.
```

OBJC_CHANGE(tree,obj,res,cx,cy,cw,ch,new_status,re_draw)

```
tree,obj,res,cx,cy,cw,ch,new_status,re_draw: iexp
```

Diese Funktion ändert den Status eines Objekts (OB_STATE) und invertiert gegebenenfalls den Teil des Objekts, der innerhalb des Clip-Bereichs liegt. Normalerweise wird OB_STATE direkt verändert und danach OBJC_DRAW aufgerufen.

```
rueck        0 = Fehler.
tree         Adresse des entsprechenden Objektbaumes.
obj          Nummer des zu ändernden Objekts.
res          Reserviert, immer 0.
cx           Linke x-Koordinate.
cy           Obere y-Koordinate.
cw           Breite und
ch           Höhe des Begrenzungsrechtecks.
new_status   Neuer Objektstatus(siehe OB_STATE).
re_draw      0 = Objekt nicht neu zeichnen.
             1 = Objekt neu zeichnen.
```

Formular-Bibliothek (form library)

Enthält Routinen zur Verwaltung von Formularen.

FORM_DO(tree,start_obj)

```
tree,star_obj: iexp
```

Diese Funktion übernimmt die vollständige Verwaltung eines Formularobjekts, bis ein Objekt mit EXIT- oder TOUCH_EXIT-Status angeklickt wurde.

rueck	Nummer des Objekts, welches zur Beendigung angeklickt wurde (bei Doppelklick wird das 15. Bit gesetzt).
tree	Adresse des entsprechenden Objektbaumes.
start_obj	Nummer des ersten Objekts im Objektbaum.

FORM_DIAL(flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y,ma_w,ma_h)

```
flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y,ma_w,ma_h: iexp
```

Diese Funktion dient der Reservierung oder Freigabe von rechteckigen Bildschirmausschnitten und zum Zeichnen von schrumpfenden oder sich ausdehnenden Rechtecken.

rueck	0, wenn Fehler aufgetreten.
flag	Funktionstyp:
	0 = FMD_START Reserviert Bildschirmbereich.
	1 = FMD_GROW Zeichnet ein sich ausdehnendes Rechteck.
	2 = FMD_SHRINK Zeichnet ein schrumpfendes Rechteck.
	3 = FMD_FINISH Gibt reservierten Bildschirmbereich wieder frei.
mi_x mi_y mi_w mi_h	Koordinaten des Rechtecks in seiner geringsten Ausdehnung.
ma_x ma_y ma_w ma_h	Koordinaten des Rechtecks in seiner größten Ausdehnung.

Erläuterung: FORM_DIAL(1,0,0,0,0,100,100,300,100) zeichnet ein sich ausdehnendes Rechteck. 0,0,0,0 bedeutet dabei, daß das Rechteck aus der Mitte des Zielrechtecks erscheint.

FORM_ALERT(button,string$)

```
button: iexp
string$: sexp
```

Zeigt eine allgemeine Warnmeldung (ALERT_BOX) an.

```
rueck    Nummer des Knopfes, mit dem die Alert-Box verlassen wurde.
button   Nummer des Default-Knopfes:
          0 = Keiner
          1 = Erster
          2 = Zweiter
          3 = Dritter
string$  Adresse des Strings, der die Meldung enthält.
         Der String hat folgendes Format:

         [i][meldung][button]

         dabei sind:

         i            genau eine Ziffer    0 = Kein icon
                                           1 = Ausrufezeichen
                                           2 = Fragezeichen
                                           3 = Stop-Schild
         meldung      Höchstens 5 Textzeilen mit jeweils 30 Zeichen,
                      voneinander durch | getrennt.
         button       Höchstens 3 durch | getrennte Button-Namen.
```

Beispiel:

```
~FORM_ALERT(1,ERR$(100))
```

FORM_ERROR(err)

```
err: iexp
```

Zeigt eine Warnmeldung für MS-DOS-Fehler an.

```
rueck   Nummer des Buttons, mit dem die Warnmeldung verlassen
        wurde.
err     MS-DOS-Fehlernummer.
```

FORM_CENTER(tree,fx,fy,fw,fh)

```
tree: iexp
fx,fy,fw,fh: ivar
```

Diese Funktion zentriert die zu übergebenden Objektkoordinaten.

rueck	Reserviert, immer 1.
tree	Adresse des entsprechenden Objektbaumes.
fx	Linke x-Koordinate.
fy	Obere y-Koordinate.
fw	Breite und
fh	Höhe des zentrierten Rechtecks.

FORM_KEYBD(tree,obj,next_obj,char,new_obj,next_char)

```
tree,obj,next_obj,char,new_obj: iexp
next_char: ivar
```

Ermöglicht Tastatureingaben in einem Formular (siehe OBJC_EDIT).

rueck	0 = Objekt mit EXIT-Status gedrückt.
	Größer 0 = Dialog noch nicht beendet.
tree	Adresse des entsprechenden Objektbaumes.
obj	Nummer des aktuellen EDIT-Objekts.
next_obj	Nummer des nächsten EDIT-Objekts.
char	Eingelesenes Zeichen.
new_obj	Neues EDIT-Objekt für den nächsten Aufruf.
next_char	Nächstes Zeichen.

Erläuterung: Diese Routine ist eine Unterroutine von FORM_DO.

FORM_BUTTON(tree,obj,clicks,new_obj)

```
tree,obj,clicks: iexp
new_obj: ivar
```

Ermöglicht Mauseingaben in einem Formular.

rueck	0 = Zuletzt wurde ein Objekt mit EXIT-Status angeklickt.
	Größer 0 = Der Dialog wurde noch nicht beendet.
tree	Adresse des entsprechenden Objektbaumes.
obj	Aktuelles Objekt.
clicks	Anzahl der Maustastendrücke.
neu_obj	Neues aktuelles Objekt.

Erläuterung: Diese Routine ist eine Unterroutine von FORM_DO.

Grafik-Bibliothek (graphics library)

GRAF_RUBBERBOX(rx,ry,min_w,min_h [last_w,last_h])

```
rx,ry,min_w,min_h: iexp
last_w,last_h: ivar
```

Die Funktion GRAF_RUBBERBOX stellt bei gedrückter Maustaste den Umriß eines Rechtecks auf dem Bildschirm dar. Dabei liegen die linke x-Koordinate und die obere y-Koordinate fest, aber die diagonal gegenüberliegende Ecke sowie Breite und Höhe des Rechtecks verändern sich je nach Maus-Cursor-Position. Der Funktionsaufruf sollte nur bei gedrückter Maustaste erfolgen, da die Funktion beim Loslassen einer Maustaste abgebrochen wird.

```
rueck     0 = Fehler aufgetreten.
rx        Linke x-Koordinate.
ry        Obere y-Koordinate.
min_w     Kleinste Breite
min_h     und Höhe des Rechtecks.
last_w    Breite und
last_h    Höhe des Rechtecks bei Funktionsabbruch.
```

Beispiel:

```
DO
  ~EVNT_BUTTON(1,1,1,mx%,my%,bu%,kb%)
  ~GRAF_RUBBERBOX(mx%,my%,1,1,w%,h%)
  BOX mx%,my%,mx%+w%,my%+h%
LOOP UNTIL BTST(kb%,3)
```

--> Wartet auf Mausklick. Dann läßt sich ein Rechteck aufziehen (Gummiband), das anschließend gezeichnet wird. Abbruch durch Drücken von Alternate und linker Maustaste.

GRAF_DRAGBOX(iw,ih,ix,iy,rx,ry,rw,rh [last_ix,last_iy])

```
iw,ih,ix,iy,vx,vy,vw,vh: iexp
last_ix,last_iy: ivar
```

Diese Funktion dient der Bewegung eines (inneren) Rechtecks innerhalb eines größeren und feststehenden Rechtecks (Rahmenrechteck). Der Funktionsaufruf sollte nur bei gedrückter Maustaste erfolgen, da die Funktion beim Loslassen einer Maustaste abgebrochen wird.

```
rueck    0 = Fehler aufgetreten.
iw       Breite und
ih       Höhe des inneren Rechtecks.
ix       Linke x-Koordinate und
iy       obere y-Koordinate des inneren Rechtecks beim
         Funktionsaufruf.
rx       Linke x-Koordinate.
ry       Obere y-Koordinate.
rw       Breite und
rh       Höhe des Rahmenrechtecks.
last_ix  Linke x-Koordinate und
last_iy  obere y-Koordinate des inneren Rechtecks bei
         Funktionsabbruch.
```

GRAF_MOVEBOX(w,h,sx,sy,dx,dy)

```
w,h,sx,sy,dx,dy: iexp
```

Die Funktion GRAF_MOVEBOX zeichnet ein bewegbares Rechteck mit gleichbleibender Breite und Höhe.

```
rueck  0 = Fehler aufgetreten.
w      Breite des bewegbaren Rechtecks.
h      Höhe des bewegbaren Rechtecks.
sx     Linke x-Koordinate des Rechtecks bei Funktionsaufruf.
sy     Obere y-Koordinate des Rechtecks bei Funktionsaufruf.
dx     Linke x-Koordinate des Rechtecks bei Funktionsende.
dy     Obere y-Koordinate des Rechtecks bei Funktionsende.
```

Beispiel:

```
~GRAF_MOVEBOX(10,20,200,20,80,120)
```

GRAF_GROWBOX(sx,sy,sw,sh,dx,dy,dw,dh)

```
sx,sy,sw,sh,dx,dy,dw,dh: iexp
```

Diese Funktion zeichnet ein sich ausdehnendes Rechteck.

```
rueck   0 = Fehler aufgetreten.
sx      Linke x-Koordinate.
sy      Obere y-Koordinate.
sw      Breite des Rechtecks zu Funktionsbeginn.
sh      Höhe des Rechtecks zu Funktionsbeginn.
dx      Linke x-Koordinate.
dy      Obere y-Koordinate.
dw      Breite des Rechtecks bei Funktionsende.
dh      Höhe des Rechtecks bei Funktionsende.
```

Beispiel:

```
~GRAF_GROWBOX(100,100,10,10,0,0,300,180)
```

GRAF_SHRINKBOX(sx,sy,sw,sh,dx,dy,dw,dh)

```
sx,sy,sw,sh,dx,dy,dw,dh: iexp
```

Diese Funktion zeichnet ein schrumpfendes Rechteck.

```
rueck   0 = Fehler aufgetreten.
sx      Linke x-Koordinate.
sy      Obere y-Koordinate.
sw      Breite des Rechtecks bei Funktionsende.
sh      Höhe des Rechtecks bei Funktionsende.
dx      Linke x-Koordinate.
dy      Obere y-Koordinate.
dw      Breite des Rechtecks zu Funktionsbeginn.
dh      Höhe des Rechtecks zu Funktionsbeginn.
```

Beispiel:

```
~GRAF_SHRINKBOX(0,0,300,180,100,100,10,10)
```

GRAF_WATCHBOX(tree,obj,in_state,out_state)

```
tree,obj,in_state,out_state: iexp
```

Diese Funktion überwacht bei gedrückter Maustaste ein Objekt (BOX) in einem Resource-Baum. Der Status des Objekts wechselt jeweils, wenn der Mauszeiger den entsprechenden Bereich erreicht oder verläßt (normal selected/normal).

rueck	Zeigt den Status des Maus-Cursors bei Loslassen der Maustaste an, 0 = außerhalb und 1 = innerhalb des zu überwachenden Objekts.
tree	Adresse des entsprechenden Objektbaumes.
obj	Objektnummer des zu überwachenden Objekts.
in_state	Status (OB_STATE) des zu überwachenden Objekts, wenn sich der Maus-Cursor innerhalb des Objekts befindet.
out_state	Status (OB_STATE) des zu überwachenden Objekts, wenn sich der Maus-Cursor außerhalb des Objekts befindet. Die entsprechende Bit-Belegung findet sich unter OB_STATE.

GRAF_SLIDEBOX(tree,parent_obj,slider_obj,flag)

```
tree,parent_obj,slider_obj,flag: iexp
```

Diese Funktion ermöglicht die Abfrage von sogenannten Schiebereglern. Ähnlich der Funktion DRAG_BOX kann dabei der Umriß eines Rechtecks (Schieber) innerhalb eines Rahmenrechtecks verschoben werden. Jedoch kann das bewegliche Rechteck nur waagerecht bzw. senkrecht bewegt werden. Außerdem muß das bewegliche Rechteck zusätzlich ein "Kindobjekt" des Rahmenrechtecks innerhalb eines Objektbaumes sein. Der Funktionsaufruf sollte nur bei gedrückter Maustaste erfolgen, da die Funktion beim Loslassen einer Maustaste abgebrochen wird.

```
rueck        Relative Stellung des "Schiebers" innerhalb des
             übergeordneten Rahmenrechtecks:
                0 = Links bzw. oben.
             1000 = Rechts bzw. unten.
tree         Adresse des entsprechenden Objektbaumes.
parent_obj   Objektnummer des "Rahmenrechtecks".
slider_obj   Objektnummer des "Schiebers".
flag         Schieberichtung:
              0 = Waagerecht
              1 = Senkrecht
```

GRAF_HANDLE(char_w,char_h,box_w,box_h)

```
char_w,char_h,box_w,box_h: ivar
```

Liefert die Kennung der VDI-Workstation, die für AES-Aufrufe intern verwendet wird, und die Ausdehnung eines Zeichens aus dem Standardzeichensatz.

rueck	Kennung der aktuellen VDI-Workstation.
char_w	Breite eines Zeichens aus dem Standardzeichensatz in Pixeln.
char_h	Höhe eines Zeichens aus dem Standardzeichensatz in Pixeln.
box_w	Breite eines Rechtecks, das ein Standardzeichen umfaßt.
box_h	Höhe eines Rechtecks, das ein Standardzeichen umfaßt.

GRAF_MOUSE(m_form,muster_adr)

```
m_form,muster_adr: iexp
```

Diese Funktion ermöglicht die Wahl der Maus-Cursor-Form. Dabei stehen acht vordefinierte oder eine vom Benutzer definierte Form zur Auswahl bereit. Komfortabler ist jedoch die Verwendung des DEFMOUSE-Befehls.

```
rueck          0 = Fehler aufgetreten.

m_form         Nummer der Maus-Cursor-Form:
                 0 = ARROW          Pfeil
                 1 = TEXT_CRSR      Senkrechter Strich
                 2 = HOURGLASS      Biene
                 3 = POINT_HAND     Zeigende Hand
                 4 = FLAT_HAND      Flache Hand
                 5 = THIN_CROSS     Dünnes Fadenkreuz
                 6 = THICK_CROSS    Dickes Fadenkreuz
                 7 = OUTL_CROSS     Fadenkreuzumriß
               255 = USER_DEF       Benutzer definiert
               256 = M_OFF          Maus-Cursor ausschalten.
               257 = M_ON           Maus-Cursor einschalten.

muster_adr     Adresse, ab der die Bit-Informationen für die
               selbstdefinierte Maus-Cursor-Form liegen. Dabei werden 37
               words wie folgt erwartet:
                 1 =          x-Koordinate und
                 2 =          y-Koordinate des Aktionspunktes.
                 3 =          Anzahl der Farbebenen, immer 1.
                 4 =          Maskenfarbe, immer 0.
                 5 =          Maus-Cursor-Farbe, immer 1.
                 6 bis 21 = Maskenform.
                22 bis 37 = Maus-Cursor-Form.
```

GRAF_MKSTATE(mx,my,m_state,k_state)

```
mx,my,m_state,k_state: iexp
```

Diese Funktion liefert die aktuellen Maus-Cursor-Koordinaten sowie Maustasten- und Tastaturstatus.

Dies ist eine AES-Routine zur Mausabfrage. Im Gegensatz zu MOUSEX usw. hält das Programm an, wenn der Pfeil in der Menüzeile steht.

```
rueck     Reserviert, immer 1.
mx        Aktuelle x-Koordinate und
my        y-Koordinate des Maus-Cursors.
m_state   Maustastenstatus:
           Bit 0 = Linke Maustaste.
           Bit 1 = Rechte Maustaste.
k_state   Tastaturstatus mit folgender Bit-Belegung:
          (Bit gesetzt = Taste gedrückt)
            1 = Rechte Shift-Taste
            2 = Linke Shift-Taste
            4 = Control-Taste
            8 = Alternate-Taste
```

Scrap-Bibliothek (scrap library)

Enthält Routinen, die den Datenaustausch zwischen verschiedenen Applikationen ermöglichen.

SCRP_READ(pfad$)

```
pfad$: svar
```

Diese Funktion liest aus dem Textpuffer, der zur Kommunikation nacheinander ausgeführter GEM-Programme dient. Gedacht ist dies vor allem für die Ablage von Ordner- und Dateinamen.

```
rueck        0 = Fehler aufgetreten.
pfad$        Zugriffspfad für das Clipboard.
```

SCRP_WRITE(pfad$)

```
pfad$: sexp
```

Diese Funktion schreibt etwas in das Scrap-Directory.

```
rueck   0 = Fehler aufgetreten.
pfad$   Neuer Zugriffspfad.
```

Beispiel:

```
~SCRP_WRITE("C:\TMP\A.X")
...
...
...
~SCRP_READ(a$)
PRINT a$
```

Dateiauswahl-Bibliothek (file selector library)

Ermöglicht die Auswahl einer Datei aus einem angezeigten Verzeichnis oder direkt durch Angabe des Zugriffspfades.

FSEL_INPUT(path$,name$,[button])

```
path$,name$: svar
button: ivar
```

Diese Funktion ruft die Standard-Dateiauswahl-Box auf. Sie entspricht dem Filselector-Befehl. Dort wird allerdings selbständig aus Pfad- und Dateinamen die vollständige Dateibezeichnung zusammengebastelt. Dies kann hier mit RINSTR(path$,"\") selbst gemacht werden.

```
rueck     0 = Fehler aufgetreten.
path$     Voreingestellter Zugriffspfad; enthält nach
          Funktionsabbruch den vom Benutzer angegebenen
          Zugriffspfad.
name$     Voreingestellter Dateiname; enthält nach
          Funktionsabbruch den vom Benutzer angegebenen
          Dateinamen.
button    Button-Status:
           0 = "Abbruch"-Button gewählt.
           1 = "Ok"-Button gewählt.
```

Fenster-Bibliothek (window library)

Übernimmt die Verwaltung von Fenstern.

WIND_CREATE(attr,wx,wy,ww,wh)

```
attr,wx,wy,ww,wh: iexp
```

Diese Funktion meldet ein neues Fenster an und legt die Fensterattribute und seine maximale Größe fest. Die aktuelle Fenster-Kennung (window handle) wird von der Funktion zurückgegeben.

Vergleiche: OPENW#n,x,y,w,h,attr

```
rueck     0 = Fehler aufgetreten, sonst Fensterkennung.

attr      Fensterattribute mit folgender Belegung:          Bit-Nr.
          &H0001   NAME      Titelbalken mit Name.            0
          &H0002   CLOSE     Schließfeld.                     1
          &H0004   FULL      Feld für volle Größe.            2
          &H0008   MOVE      Bewegungsbalken.                 3
          &H0010   INFO      Info-Zeile.                      4
          &H0020   SIZE      Einstellung der Fenstergröße.    5
          &H0040   UPARROW   Pfeil nach oben.                 6
          &H0080   DNARROW   Pfeil nach unten.                7
          &H0100   VSLIDE    Senkrechter Schieber.            8
          &H0200   LFARROW   Pfeil nach links.                9
          &H0400   RTARROW   Pfeil nach rechts.              10
          &H0800   HSLIDE    Waagerechter Schieber.          11

wx        Maximale linke x-Koordinate.
wy        Obere y-Koordinate.
ww        Breite des Fensters.
wh        Höhe des Fensters.
```

WIND_OPEN(handle,wx,wy,ww,wh)

```
handle,wx,wy,ww,wh: iexp
```

Diese Funktion stellt ein zuvor mit WIND_CREATE erzeugtes Fenster auf dem Bildschirm dar.

Vergleiche: OPENW

```
rueck   0 = Fehler aufgetreten.
handle  Kennung des Fensters.
wx      Linke x-Koordinate.
wy      Obere y-Koordinate.
ww      Breite und
wh      Höhe des Fensters zu Beginn der Funktion.
```

WIND_CLOSE(handle)

```
handle: iexp
```

Diese Funktion stellt das Gegenstück zu WIND_OPEN dar und schließt das entsprechende Fenster.

Vergleiche: CLOSEW

```
rueck   0 = Fehler aufgetreten.
handle  Kennung des entsprechenden Fensters.
```

WIND_DELETE(handle)

```
handle: iexp
```

Diese Funktion löscht ein Fenster und gibt den reservierten Speicherplatz sowie die entsprechende Fensterkennung wieder frei.

Vergleiche: CLOSEW

```
rueck   0 = Fehler aufgetreten.
handle  Kennung des entsprechenden Fensters.
```

WIND_GET(handle,field,w1,w2,w3,w4)

```
handle,field: iexp
w1,w2,w3,w4: ivar
```

Diese Funktion liefert je nach Funktionsnummer Informationen über ein Fenster.

rueck 0 = Fehler aufgetreten.
handle Kennung des entsprechenden Fensters.

field In Abhängigkeit von der Funktionsnummer werden Informationen über das entsprechende Fenster in wi_gw1, wi_gw2, wi_gw3 und wi_gw4 zurückgegeben.

4 WX_WORKXYWH berechnet die Koordinaten des Arbeitsbereiches:
   w1 Linke x-Koordinate.
   w2 Obere y-Koordinate.
   w3 Breite und
   w4 Höhe des entsprechenden Fensters.

5 WF_CURRXYWH berechnet die Koordinaten des gesamten Fensters einschließlich der Ränder:
   w1 Linke x-Koordinate.
   w2 Obere y-Koordinate.
   w3 Breite und
   w4 Höhe des entsprechenden Fensters.

6 WF_PREVXYWH berechnet die Gesamtgröße des vorherigen Fensters:
   w1 Linke x-Koordinate.
   w2 Obere y-Koordinate.
   w3 Breite und
   w4 Höhe des entsprechenden Fensters.

7 WF_FULLXYWH berechnet die Gesamtgröße des Fenster in seiner größtmöglichen Ausdehnung:
- w1 Linke x-Koordinate.
- w2 Obere y-Koordinate.
- w3 Breite und
- w4 Höhe des entsprechenden Fensters.

8 WF_HSLIDE liefert die Stellung des waagerechten Schiebers:
- w1 (1 = ganz links, 1000 = ganz rechts)

9 WF_VSLIDE liefert die Stellung des senkrechten Schiebers:
- w1 (1 = ganz oben, 1000 = ganz unten)

10 WF_TOP liefert die Kennung des obersten Fensters:
- w1 Fensterkennung

11 WF_FIRSTXYWH liefert die Koordinaten des ersten Rechtecks innerhalb der Rechteckliste des entsprechenden Fensters:
- w1 Linke x-Koordinate.
- w2 Obere y-Koordinate.
- w3 Breite des entsprechenden Fensters.
- w4 Höhe des entsprechenden Fensters.

12 WF_NEXTXYWH liefert die Koordinaten des nächsten Rechtecks innerhalb der Rechteckliste des entsprechenden Fensters:
- w1 Linke x-Koordinate.
- w2 Obere y-Koordinate.
- w3 Breite des entsprechenden Fensters.
- w4 Höhe des entsprechenden Fensters.

13 WF_RESVD ist reserviert.

15 WF_HSLIZE berechnet die Größe des waagerechten Schiebers relativ zum Balken:

w1 -1 = Minimalgröße
(1 = klein, 1000 = gesamte Breite)

16 WF_VSLIZE berechnet die Größe des senkrechten Schiebers relativ zum Balken:

w1 -1 = Minimalgröße
(1 = klein, 1000 = gesamte Breite)

WIND_SET(handle,field,w1,w2,w3,w4)

```
handle,field: iexp
w1,w2,w3,w4: ivar
```

Diese Funktion verändert je nach Funktionsnummer Teile des entsprechenden Fensters.

```
rueck     0 = Fehler aufgetreten.
handle    Kennung des entsprechenden Fensters.

field     In Abhängigkeit von der Funktionsnummer werden
          Teile des entsprechenden Fensters verändert:

             1   WF_KIND  setzt neue Fensterteile
                          (wie unter WIND_CREATE festgelegt).
                  w1   Neues Fensterteil.

             2   WF_NAME  legt einen neuen Fenstertitel fest:
                  w1   und
                  w2   enthalten die Adresse des Strings.

             3   WF_INFO  legt eine neue Info-Zeile fest:
                  w1   und
                  w2   enthalten die Adresse des Strings.

             5   WF_CURRXYWH legt die Fenstergröße fest:
                  w1   Linke x-Koordinate.
                  w2   Obere y-Koordinate.
                  w3   Breite des entsprechenden Fensters.
                  w4   Höhe des entsprechenden Fensters.

             8   WF_HSLIDE positioniert den waagerechten Schieber:
                  w1   (1 = ganz links, 1000 = ganz rechts)

             9   WF_VSLIDE positioniert den senkrechten Schieber:
                  w1   (1 = ganz oben, 1000 = ganz unten)
```

10 WF_TOP legt das oberste (aktuelle) Fenster fest.

14 WF_NEWDESK legt neuen Desktop-Menübaum fest:
- w1 High-Byte.
- w2 Low-Byte der Adresse des Menübaumes.
- w3 Nummer des ersten zu zeichnenden Objekts.

15 WF_HSLIZE legt die Größe des waagerechten Schiebers relativ zum Balken fest:
- w1 -1 = Minimalgröße
  (1 = klein, 1000 = gesamte Breite)

16 WF_VSLIZE legt die Größe des senkrechten Schiebers relativ zum Balken fest:
- w1 -1 = Minimalgröße
  (1 = klein, 1000 = gesamte Breite)

WIND_FIND(fx,fy)

```
fx,fy: iexp
```

Diese Funktion ermittelt die Kennung eines Fensters an anzugebenden Koordinaten.

rueck	0 = kein Fenster mit diesen Koordinaten, sonst Kennung des gefundenen Fensters.
fx	x-Koordinate der entsprechenden Bildschirmposition.
fy	y-Koordinate der entsprechenden Bildschirmposition.

WIND_UPDATE(flag)

```
flag: iexp
```

Diese Funktion koordiniert alle Maßnahmen, die mit dem Bildschirmaufbau zusammenhängen, insbesondere in Verbindung mit Pull-Down-Menüs.

```
rueck        0 = Fehler aufgetreten.
flag         Funktionsnummer:
              0 = END_UPDATE   Bildschirmaufbau abgeschlossen.
              1 = BEG_UPDATE   Bildschirmaufbau beginnt.
              2 = END_MCTRL    Applikation gibt Mauskontrolle ab.
              3 = BEG_MCTRL    Applikation übernimmt die volle Maus-
                               Kontrolle, dabei sind insbesondere die
                               Pull-Down-Menüs gesperrt.
```

WIND_CALC(w_type,attr,ix,iy,iw,ih,ox,oy,ow,oh)

```
w_type,attr,ix,iy,iw,ih: iexp
ox,oy,ow,oh: ivar
```

Diese Funktion berechnet die Gesamtausdehnung eines Fensters aus der Größe des Arbeitsbereichs oder umgekehrt die Größe des Arbeitsbereichs aus der Größe des Fensters.

rueck 0 = Fehler aufgetreten.

w_type 0 = Gesamtausdehnung berechnen.
1 = Arbeitsbereich berechnen.

attr Fensterteile:

			Bit-Nr.
&H0001	NAME	Titelbalken mit dem Namen.	0
&H0002	CLOSE	Schließfeld.	1
&H0004	FULL	Feld für volle Größe.	2
&H0008	MOVE	Bewegungsbalken.	3
&H0010	INFO	Info-Zeile.	4
&H0020	SIZE	Einstellung der Fenstergröße.	5
&H0040	UPARROW	Pfeil nach oben.	6
&H0080	DNARROW	Pfeil nach unten.	7
&H0100	VSLIDE	Senkrechter Schieber.	8
&H0200	LFARROW	Pfeil nach links.	9
&H0400	RTARROW	Pfeil nach rechts.	10
&H0800	HSLIDE	Waagerechter Schieber.	11

ix
iy
iw
ih Bekannte Koordinaten.

ox
oy
ow
oh Errechnete Koordinaten.

Resource-Bibliothek (resource library)

Enthält Routinen zur Erstellung einer grafischen Benutzeroberfläche, die unabhängig von der gewählten Auflösung den Datenaustausch zwischen Benutzer und Programm vereinfacht.

RSRC_LOAD(name$)

```
name$: sexp
```

Diese Funktion reserviert Speicherplatz und lädt eine Resource-Datei. Anschließend werden interne Zeiger gesetzt und Koordinaten von Zeichen- in Pixelformat umgerechnet. Für Resourcen, die man selbst im Speicher definiert, muß dazu RSRC_OBFIX verwendet werden.

```
rueck        0 = Fehler aufgetreten.
name$        Zugriffspfad der entsprechenden Resource-Datei.
```

Beispiel:

```
~RSRC_LOAD("TEST.RSC")
```

RSRC_FREE()

Diese Funktion gibt den durch RSRC_LOAD reservierten Speicherplatz wieder frei.

```
rueck   0 = Fehler aufgetreten.
```

Beispiel:

```
~RSRC_FREE()
```

RSRC_GADDR(type,index,adr)

```
type,index: iexp
adr: ivar
```

Diese Funktion ermittelt die Adresse einer Resource-Struktur nach dem Laden durch RSRC_LOAD. Diese Funktion wird normalerweise nur für Bäume und Alerts (ad_frstr) eingesetzt.

```
rueck        0 = Fehler aufgetreten.

type         Gesuchter Strukturtyp:
              0   Objektbaum
              1   OBJECT
              2   TEDINFO
              3   ICONBLK
              4   BITBLK
              5   STRING
              6   imagedata
              7   obspec
              8   te_ptext
              9   te_ptmplt
             10   te_pvalid
             11   ib_pmask
             12   ib_pdata
             13   ib_ptext
             14   bi_pdata
             15   ad_frstr
             16   ad_frimg

index        Objektnummer der gesuchten Struktur.
adr          Adresse der gesuchten Struktur.
```

Beispiel:

```
~RSRC_GADDR(0,0,TREE%)
```

RSRC_SADDR(type,index,adr)

```
type,index: iexp
adr: ivar
```

Diese Funktion setzt Adressen in Resourcen.

rueck 0 = Fehler aufgetreten.

type
- 0 Objektbaum.
- 1 OBJECT
- 2 TEDINFO
- 3 ICONBLK
- 4 BITBLK
- 5 STRING
- 6 imagedata
- 7 obspec
- 8 te_ptext
- 9 te_ptmplt
- 10 te_pvalid
- 11 ib_pmask
- 12 ib_pdata
- 13 ib_ptext
- 14 bi_pdata
- 15 ad_frstr
- 16 ad_frimg

index Position innerhalb der Datenstruktur, an die re_saddr geschrieben wird.

adr Die abzuspeichernde Adresse.

RSRC_OBFIX(tree,obj)

```
tree,obj: iexp
```

Diese Funktion wandelt die Objektkoordinaten von Zeichen- in Pixeldarstellung um. Das ist notwendig, wenn Objekte nicht mit RSRC_LOAD geladen, sondern direkt in das Programm integriert sind.

rueck	Reserviert, immer 1.
tree	Adresse des entsprechenden Objektbaumes.
obj	Objektnummer des umzuwandelnden Objekts.

Shell-Bibliothek (shell library)

Umfaßt Routinen, die den Aufruf einer Applikation durch ein anderes Programm ermöglichen, wobei die Applikation und deren Umgebung erhalten bleiben.

SHEL_READ(cmd_string$,tail_string$)

```
cmd_string,tail_string: svar
```

Diese Funktion liefert den Namen und die Kommandozeile, mit der ein Programm vom Desktop aufgerufen wurde (Kommandozeilen durch Anwendung anmelden).

```
rueck           0 = Fehler aufgetreten.
cmd_string$     String für die Kommandozeile.
tail_string$    String für den Namen.
```

SHEL_WRITE(prg,grf,gem,cmd$,num$)

```
prg,grf,gem: iexp
cmd$,num$: sexp
```

Diese Funktion teilt den AES mit, daß eine andere Applikation gestartet werden soll. Anders als beim GEMDOS-Befehl pexec bleibt das aktuelle Programm jedoch nicht speicherresident.

rueck	0 = Fehler aufgetreten.
prg	0 = Zurück zum Desktop.
	1 = Neues Programm laden.
grf	0 = TOS-Programm.
	1 = Grafik-Applikation.
gem	0 = Keine GEM-Applikation.
	1 = GEM-Applikation.
cmd$	String mit der Kommandozeile.
num$	String mit dem Namen.

Beispiel:

```
~ SHEL_WRITE(1,1,1,"","GFABASIC.PRG")
```

--> Dies bedeutet, daß nach Verlassen des BASIC und Rückkehr zum Desktop wieder das GFA-BASIC gestartet wird.

SHEL_GET(anz,x$)

```
anz: iexp
x$: svar
```

Diese Funktion ermöglicht das Lesen aus dem Speicher (desktop.inf).

rueck	0 = Fehler aufgetreten.
anz	Anzahl der zu lesenden Bytes.
x$	String zur Aufnahme der gelesenen Zeichen.

SHEL_PUT(len,x$)

```
len: iexp
x$: sexp
```

Diese Funktion ermöglicht das Schreiben einer anzugebenden Anzahl von Zeichen in den Environment-Speicher des GEM (desktop.inf).

rueck	0 = Fehler aufgetreten.
x$	String, der die zu schreibenden Zeichen enthält.
len	Anzahl der zu schreibenden Bytes.

Beispiel:

```
' GFA-BASIC anmelden
~SHEL_GET(2000,a$)
q%=INSTR(a$,CHR$(26))
IF q%
  a$=LEFT$(a$,q%-1)
  IF INSTR(a4,"GFABASIC.PRG")=0
    a$=a$+"#G 03 04   c:\GFA\GFABASIC.PRG@ *.GFA@ "+MKI$(&hd0a)+CHR$(26)
   ~SHEL_PUT(LEN(a$),a$)
  ENDIF
ENDIF
```

--> Das Programm meldet, daß C:\GFA\GFABASIC.PRG an, sodaß bei jedem Doppelklick auf Files mit der Extension .GFA dieses GFA-BASIC gestartet wird.

Anmerkung: Arbeit sichern mit:

```
~SHEL_GET(3000,a$)
OPEN"O",#1,"C:\DESKTOP.INF"
PRINT #1,LEFT$(A$,INSTR(A$,CHR$(26))
CLOSE #1
' Wichtig ist das CHR$(26)
```

SHEL_FIND(pfad$)

```
pfad$: svar
```

Diese Funktion sucht eine Datei im aktuellen Ordner, im Wurzelverzeichnis des aktiven Laufwerkes und des Laufwerkes A: und gibt im Erfolgsfall den vollständigen Pfad zu dieser Datei aus (entweder xx.x oder \xx.x oder a:\xx.x).

rueck	0 = Fehler aufgetreten.
pfad$	String, der den Dateinamen der gesuchten Datei enthält. Nach Funktionsende enthält dieser String den vollständigen Zugriffspfad auf die Datei.

SHEL_ENVRN(adr,such$)

```
adr: avar (32 Bit)
such$: sexp
```

Diese Funktion dient der Bestimmung von Variablenwerten im GEM-Environment.

```
rueck        Reserviert, immer 1.
adr          Adresse, vor der der in such$ angegebene String steht.
such$        String, der den Such-Parameter enthält (z.B. A:\).
```

Beispiel:

```
PRINT SHEL_ENVRN(a%,"P")
PRINT CHAR{a%-1}
```

--> Ausgabe: PATH=;A:\

Beispielprogramme

Im letzten Abschnitt folgen Beispielprogramme zum Thema Grafik-Bibliothek, Dialogboxen, Menüleisten- und Fensterprogrammierung.

```
'
' Grafik-Bibliothek
'
REPEAT
  CLS
  PRINT "rubber = 1, drag = 2, move = 3, grow_shrink = 4, ende = 0"
  INPUT "wahl: ";wahl%
  ON wahl% GOSUB rubber,drag,move,grow_shrink
UNTIL wahl%=0
'
EDIT
'
PROCEDURE rubber
  GRAPHMODE 3
  DEFFILL 1,2,4
  REPEAT
    MOUSE mx%,my%,mk%
    IF mk% AND 1
      x1%=mx%
      y1%=my%
      ~GRAF_RUBBERBOX(x1%,y1%,16,16,lx%,ly%)
      PBOX x1%,y1%,x1%+lx%,y1%+ly%
    ENDIF
  UNTIL mk% AND 2
RETURN
'
PROCEDURE drag
  GRAPHMODE 1
  DEFFILL 1,2,4
  REPEAT
    MOUSE mx%,my%,mk%
```

```
    BOX 50,50,200,150
    BOX 40,40,400,300
    IF mk% AND 1
      ~GRAF_DRAGBOX(150,100,50,50,40,40,400,300,lx%,ly%)
      PBOX lx%,ly%,lx%+150,ly%+100
    ENDIF
  UNTIL mk% AND 2
  RETURN
  '
  PROCEDURE move
    GRAPHMODE 1
    DEFFILL 1,2,4
    b%=64
    h%=64
    FOR i%=0 TO 639-b% STEP b%
      FOR j%=0 TO 399-h% STEP h%
        ~GRAF_MOVEBOX(b%,h%,i%,j%,639-i%,399-j%)
      NEXT j%
    NEXT i%
  RETURN
  '
  PROCEDURE grow_shrink
    GRAPHMODE 1
    ~GRAF_GROWBOX(319,199,16,16,0,0,639,399)
    PRINT "Das war graf_growbox"
    PRINT "Bitte eine Taste drücken"
    ~INP(2)
    ~GRAF_SHRINKBOX(319,199,16,16,0,0,639,399)
    PRINT "Das war graf_shrinkbox"
    PRINT "Bitte eine Taste drücken"
    ~INP(2)
  RETURN
```

--> Fragt nach der gewünschten Routine und verzweigt in die entsprechende Prozedur. Wird eine 1 angegeben, erscheint bei gedrückter linker Maustaste der Umriß eines Rechtecks auf dem Bildschirm. Dabei liegt die linke x-Koordinate und die obere y-Koordinate fest, aber die diagonal gegenüberliegende Ecke sowie Breite und Höhe des Rechteckes verändern sich je nach der aktuellen Maus-Cursor-Position.

Läßt man die Maustaste los, wird ein gefülltes Rechteck mit den Koordinaten des zuletzt dargestellten Rechtecks gezeichnet. Die Prozedur wird durch Drücken der rechten Maustaste verlassen.

Bei Angabe einer 2 wird eine große Rahmenbox gezeichnet, in der ein kleineres Rechteck enthalten ist. Bei gedrückter linker Maustaste kann das innere Rechteck innerhalb des Rahmenrechtecks verschoben ("gedragt") werden. Nach dem Loslassen der Maustaste wird ein gefülltes Rechteck mit den Koordinaten des zuletzt gezeichneten inneren Rechtecks dargestellt. Die Prozedur wird durch Drücken der rechten Maustaste verlassen.

Durch die Wahl von 3 wird in eine Prozedur verzweigt, die GRAF_MOVEBOX aufruft, dabei wird eine Reihe von Rechtecken dargestellt, die sich kreisförmig um den Bildschirmmittelpunkt bewegt.

Schließlich wird bei der Angabe von 4 ein sich ausdehnendes Rechteck gezeichnet, auf einen Tastendruck gewartet und anschließend ein schrumpfendes Rechteck dargestellt. Das Programm wird duch die Angabe von 0 abgebrochen.

```
' Dialogbox-Verwaltung
'
DIM r%(3)
'           Resource-Indizes für DIALOG
form1%=0    !Formular/Dialog
ikon1%=1    !ICON in Baum FORM1
vor%=2      !FTEXT in Baum FORM1
nac%=3      !FTEXT in Baum FORM1
str%=4      !FTEXT in Baum FORM1
ort%=5      !FTEXT in Baum FORM1
abbruch%=6  !BUTTON in Baum FORM1
ok%=7       !BUTTON in Baum FORM1
r%(1)=8     !BUTTON in Baum FORM1
r%(2)=9     !BUTTON in Baum FORM1
r%(3)=10    !BUTTON in Baum FORM1
ausgabe%=11 !STRING in Baum FORM1
'
~RSRC_FREE()
RESERVE FRE(0)+32000
laden_ok!=RSRC_LOAD("\dialog.rsc") !Resource laden
IF NOT laden_ok!
  ALERT 1,"RSC nicht gefunden",1," Return ",a%
  RESERVE FRE(0)+32000
  EDIT
ENDIF
~RSRC_GADDR(0,0,tree%)              !Adresse des Objektbaumes ermitteln
~FORM_CENTER(tree%,x%,y%,w%,h%)     !Objektbaumkoordinaten zentrieren
'
' Texte in den Edit-Feldern vorbelegen
CHAR{{OB_SPEC(tree%,vor%)}}="Johann Sebastian"
CHAR{{OB_SPEC(tree%,nac%)}}="Bach"
CHAR{{OB_SPEC(tree%,str%)}}="Kantatengasse 77"
CHAR{{OB_SPEC(tree%,ort%)}}="Dresden"
'
~OBJC_DRAW(tree%,0,1,x%,y%,w%,h%)  !Objektbaum zeichnen
'
```

```
REPEAT
  ex%=FORM_DO(tree%,0)      !Objekt mit Exit-Status angeklickt ?
  '
  ' Texte aus den Edit-Feldern in entsprechende Strings einlesen
  vorname$=CHAR{{OB_SPEC(tree%,vor%)}}
  nachname$=CHAR{{OB_SPEC(tree%,nac%)}}
  strasse$=CHAR{{OB_SPEC(tree%,str%)}}
  ort$=CHAR{{OB_SPEC(tree%,ort%)}}
  '
  FOR i%=1 TO 3
    IF BTST(OB_STATE(tree%,r%(i%)),0)   !welcher Radio-Button
      radio%=r%(i%)                     !wurde angeklickt?
    ENDIF
  NEXT i%
UNTIL ex%=ok% OR ex%=abbruch%
'
~RSRC_FREE()   !reservierten Speicherplatz wieder freigeben
RESERVE FRE(0)+32000
'
CLS
PRINT "Ende mit : ";ex%
PRINT "Vorname  : ";vorname$
PRINT "Nachname : ";nachname$
PRINT "Strasse  : ";strasse$
PRINT "Ort      : ";ort$
PRINT "Radio    : ";radio%
```

--> Die Resource-Datei "DIALOG.RSC" wird geladen und die Anfangsadresse des Objektbaumes ermittelt sowie die Objektbaum-Koordinaten zentriert. Danach werden Default-Strings in die Edit-Felder geschrieben. Das geschieht mit Hilfe der Funktion OB_SPEC.

Diese Funktion liefert einen Zeiger auf eine TEDINFO-Struktur, in der verschiedene Informationen zu einem Text enthalten sind, z.B. die Adresse des Textes und der zugehörigen Schablone, die Länge usw. Diese TEDINFO-Struktur enthält wiederum einen Zeiger; und zwar auf den eigentlichen String. So stellt sich der Weg vom Objekt zum String wie folgt dar:

Objekt --> TEDINFO-Struktur --> String

OB_SPEC(tree%,vor%) liefert einen Zeiger auf eine TEDINFO-Struktur.

{OB_SPEC(tree%,vor%)} liefert den Zeiger, auf den die TEDINFO-Struktur zeigt.

CHAR{{OB_SPEC(tree%,vor%)}} liefert die im Resource-Construction-Set eingestellte Zeichenkette, in diesem Fall: 01234567890123456789 01.

CHAR{{OB_SPEC(tree%,vor%)}} = "Johann Sebastian" weist dieser voreingestellten Zeichenkette einen neuen Wert (nämlich Johann Sebastian) zu.

Anschließend wird der Objektbaum gezeichnet und innerhalb der REPEAT-UNTIL-Schleife mittels der FORM_DO Routine abgefragt. Dabei wird bei jedem Durchgang geprüft, ob ein Objekt mit Exit-Status angeklickt wurde und, falls ja, die Schleife verlassen. Als Abbruchkriterium dient der Vergleich des Rückgabewertes von FORM_DO mit den Objektnummern von Abbruch und Ok-Button.

Innerhalb der FOR-NEXT-Schleife wird mit Hilfe von BTST der Objektstatus der Radio-Buttons überprüft. Dazu wurden die Objektnummern der Radio-Buttons der Einfachheit halber zu Programmbeginn in ein Integer-Array gelesen (r%(1) bis r%(3)), was die simple Abfrage über die FOR-NEXT-Schleife ermöglicht. OB_STATE liefert den Objektstatus (siehe auch zu Beginn dieses Kapitels) und BTST überprüft, ob das nullte Bit des entsprechenden Radio-Buttons gesetzt ist oder nicht (also ob das Objekt angewählt wurde oder nicht). Da immer nur ein Radio-Button aus einer Gruppe gleichzeitig angewählt werden kann, wird die Objektnummer des entsprechenden Radio-Buttons in eine Variable eingelesen, so daß bei Abbruch der Abfrage die Objektnummer des zuletzt gewählten Radio-Buttons in dieser Variable steht.

Schließlich wird mit RSRC_FREE der für das Resource reservierte Speicherplatz wieder freigegeben (wichtig !!, sonst schrumpft der zur Verfügung stehende Speicherplatz bei jedem Programmstart, was bald zum Absturz des Rechners führt).

Am Schluß wird die Nummer des Objekts angegeben, mit dem die FORM_DO Routine verlassen wurde (6 = Abbruch oder 7 = Ok). Danach folgen die Zeichenketten, die bei Verlassen der Routine in den vier Edit-Feldern standen, und die Objektnummer des zuletzt angewählten Radio-Buttons.

```
' Menüleisten-Programmierung
'
' Speicherplatz für Resource reservieren, Resource laden
' und Menüleiste darstellen.
RESERVE FRE(0)-33000
IF RSRC_LOAD("\handbuch.rsc")=0
  ALERT 1,"Resource-Datei|nicht gefunden.",1," Return ",a%
  RESERVE FRE(0)+33000
  EDIT
ENDIF
~RSRC_GADDR(0,0,menu_adr%)
~MENU_BAR(menu_adr%,1)
'
' Message-Buffer einrichten und nützliche Variablen vorbereiten
DIM message_buffer%(3)
mes_adr%=V:message_buffer%(0)
ABSOLUTE mes_type&,mes_adr%
ABSOLUTE m_titel&,mes_adr%+6
ABSOLUTE m_eintrag&,mes_adr%+8
'
REPEAT
  ~EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,mes_adr%,100)
  ' falls ein Menüpunkt gewählt wurde
  IF mes_type&=10
    ' falls ein anderer Eintrag als beim letzten Mal gewählt wurde
    IF obj_nr%<>m_eintrag&
      obj_nr%=m_eintrag&
      titel$=CHAR{OB_SPEC(menu_adr%,m_titel&)}
      eintrag$=CHAR{OB_SPEC(menu_adr%,m_eintrag&)}
      PRINT AT(3,20);"Menütitel:    ";titel$;SPC(15)
      PRINT AT(3,22);"Menüeintrag:   ";eintrag$;SPC(15)
      ~MENU_TNORMAL(menu_adr%,m_titel&,1)
    ENDIF
  ENDIF
UNTIL MOUSEK=2
'
' Menüleiste entfernen, Resource aus Speicher entfernen und
```

```
' reservierten Speicher wieder zurückholen.
~MENU_BAR(menu_adr%,0)
~RSRC_FREE()
RESERVE FRE(0)+33000
END
```

--> Zu Beginn des Programms werden 33 Kilobyte Speicherplatz für eine Resource-Datei reserviert. Der Speicherplatz wird am Ende des Programms wieder an den GFA-BASIC-Interpreter zurückgegeben.

Nach der Speicherplatzreservierung wird das Resource-File geladen, die Adresse des Menübaumes bestimmt und das Menü auf dem Bildschirm dargestellt.

Die Verwaltung des Menüs erfolgt mit einer EVNT_MULTI-Funktion, die den Menübaum überwacht. Falls Elemente des Menübaumes vom Benutzer angesprochen werden, so werden Meldungen in einen sogenannten Message-Puffer geschrieben. Dieser ist ein 16 Byte langer Speicherbereich, der aus 8 Worten (je 2 Byte) aufgebaut ist. Dieser Puffer wird in einem mit DIM angelegten Array eingerichtet.

In den ersten beiden Bytes dieses Puffers steht eine 10, wenn ein Menüeintrag angewählt wurde. Ab dem sechsten Byte steht in einem solchen Fall die Objektnummer des Menütitels, unter dem ein Eintrag angewählt wurde. Die Objektnummer des Eintrags steht ab dem achten Byte des Message-Puffers. Die anderen Elemente des Puffers sind im Moment nicht von Bedeutung. Um die relevanten Elemente leicht ansprechen zu können, werden ihnen mit ABSOLUTE Variablennamen zugeordnet.

Nun folgt eine Schleife, die durch das Drücken der rechten Maustaste verlassen werden kann. In ihr steht die EVNT_MULTI-Funktion, die das Menü überwacht. Falls ein Menüeintrag gewählt wurde, so stehen in den Variablen m_titel& und m_eintrag& die Objektnummern des Menütitels der heruntergeklappten Menüleiste und des gewählten Menüeintrags. Aus diesen beiden läßt sich mit Hilfe der Funktionen CHAR und OB_SPEC der Text des Menütitels und des Menüeintrags bestimmen. Dazu muß man wissen, daß OB_SPEC einen Zeiger auf eine TEDINFO-Struktur darstellt, in dessen ersten vier Bytes ein Zeiger auf den Text des Objekts steht.

OB_SPEC(...) ist also die Adresse des interessierenden Strings, der mit CHAR gelesen werden kann, weil er mit einem Null-Byte endet (siehe auch das Programm zur Dialogboxverwaltung).

Nach der Ausgabe der beiden Texte auf dem Bildschirm wird der invertierte Menütitel wieder auf den nicht invertierten Zustand zurückgeschaltet.

```
' Fensterprogrammierung
'
DEFFILL 1,2,4
PBOX 0,19,639,399
DEFFILL 1,0
'
DIM message_buffer%(3)  ! 16 Byte
adr_mes%=V:message_buffer%(0)
'
ABSOLUTE word0&,adr_mes%
ABSOLUTE x&,adr_mes%+8
ABSOLUTE y&,adr_mes%+10
ABSOLUTE w&,adr_mes%+12
ABSOLUTE h&,adr_mes%+14
'
handle&=WIND_CREATE(&X101111,0,19,639,380)
'
titel$="Window"
adr_tit%=V:titel$
~WIND_SET(handle&,2,CARD(SWAP(adr_tit%)),CARD(adr_tit%),0,0)
~WIND_OPEN(handle&,100,100,200,100)
~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
PBOX wx&,wy&,wx&+ww&,wy&+wh&
'
raus!=FALSE
REPEAT
  ' nächste Zeile nur aus Platzgründen geteilt
  ~EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,adr_mes%,100)
  SELECT word0&
  CASE 22                                  ! WM_CLOSED
    raus!=TRUE
  CASE 23                                  ! WM_FULLED
    ~WIND_SET(handle&,5,1,19,638,380)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
    word0&=0
  CASE 27,28                               ! WM_SIZED, WM_MOVED
```

```
    IF w&<100
      w&=100
    ENDIF
    IF h&<80
      h&=80
    ENDIF
    ~WIND_SET(handle&,5,x&,y&,w&,h&)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
    word0&=0
  ENDSELECT
UNTIL raus!
~WIND_CLOSE(handle&)
~WIND_DELETE(handle&)
```

--> Zu Beginn des Programms wird ein Message-Puffer eingerichtet, in den die Meldungen über angewählte Fensterrandelemente aufgenommen werden sollen. Den für dieses Programm wichtigen Elementen dieses Speicherbereichs werden mit ABSOLUTE Variablennamen zugeordnet.

Anschließend wird das Fenster mit WIND_CREATE erstellt und ihm mit WIND_SET ein Name zugewiesen. Nach dem Öffnen des Fensters mit WIND_OPEN wird die Arbeitsfläche des Fensters erfragt und diese mit einem weißen Rechteck überschrieben.

In der anschließenden Schleife wird die Auswahl der Fensterbedienelemente von einer EVNT_MULTI-Funktion überwacht. Die verschiedenen Ereignisse werden hinter den zugehörigen CASE-Anweisungen abgehandelt. Wurde das Fensterschließelement gewählt, so wird das Programm, nachdem das Fenster geschlossen und aus dem Speicher entfernt wurde, beendet.

12 - Anhang

Kompatibilität zu GFA-BASIC 2.xx

Es ist in GFA-BASIC 3.0 möglich, Programme aus älteren GFA-BASIC-Versionen zu verwenden. Zu diesem Zweck müssen diese Programme mit der alten Version als ASCII-Datei abgespeichert (Save,A) und in der neuen Version mit Merge geladen werden. Wenn diese Übertragung einmal gemacht wurde, kann das Programm nun in GFA-BASIC 3.0 mit Save und Load weiter verwendet werden.

Alle Befehle, die in älteren Versionen des Basic-Interpreters enthalten sind, finden sich auch in der Version 3.0. Es gibt aber zum Teil kleine Abweichungen in der Befehlsbedeutung, die eine Anpassung erforderlich machen.

MUL DIV

Die Befehle MUL und DIV verarbeiten in der Version 3.0 bei Integervariablen (|,&,%) auch nur Integerparameter.

In den älteren Versionen führte

```
a%=10
MUL a%,2.5
```

dazu, daß in a% der Wert 25 steht. Die Version 3.0 beachtet die Nachkommastellen bei den Parametern des MUL-Befehls jedoch nicht, so daß danach a% gleich 20 ist. Diese Inkompatibilität ist der Preis dafür, daß durch die Verwendung einer echten Integerarithmetik die genannten Befehle um ein mehrfaches schneller sind als in den älteren GFA-BASIC-Versionen. Dies gilt jedoch nur für Integervariablen.

PRINT USING

PRINT USING gibt bei Überlauf nicht mehr die Zahl unformatiert aus, sondern nur die Ziffern, die in das entsprechende Format passen und damit unter Umständen falsche Werte statt falscher Formate. Außerdem werden im Exponentialformat auch viele "^"-Zeichen korrekt behandelt und bei mehreren Vorkommastellen erfolgt eine Anpassung des Exponenten.

CLS - PRINT TAB

Bei CLS wird jetzt ESC-E-CR ausgegeben, so daß das erste PRINT TAB korrekt behandelt wird.

KEYPAD

Ein Programm, das die Tasten des Ziffernblocks mit Alternate und/oder Control abfragt, benötigt jetzt ein KEYPAD 0 um die spezielle Tastaturbelegung abzuschalten. Gleiches gilt für die Funktionstasten mit Alternate.

MOUSEX - MOUSEY

Wenn Fenster aktiv sind und MOUSEX/MOUSEY abgefragt werden, so ergeben sich oberhalb und links der Fenstergrenzen (bzw.CLIP OFFSET) negative Koordinaten. In der Version 2.X wurde CARD(MOUSEX) bzw. CARD(MOUSEY) zurückgeliefert.

Die im Interpreter erstellten Programme können anschließend durch die Verwendung des GFA-BASIC-Compilers 3.0 erheblich beschleunigt werden. Dadurch entstehen Programme, die auch ohne den Interpreter lauffähig sind. Weiterhin ist die Verwendung des im Lieferumfang enthaltenen RUN-ONLY-Interpreters möglich. Dieser kann von eingetragenen GFA-BASIC-Käufern eigenen Programmen beigelegt werden. Dadurch ist die Nutzung von GFA-BASIC-Programmen für Benutzer ohne eigenen GFA-BASIC-Interpreter möglich.

GEMDOS-Tabelle

~GEMDOS(0) **pterm0()**

Beendet ein Programm. Darf in GFA-BASIC nicht verwendet werden.

r%=GEMDOS(1) **cconin()**

Zeichen von Tastatur mit Anzeige auf dem Bildschirm lesen (vgl. INP(2)).

```
r%  Bit 0-7: ASCII-Code, 16-23: Scan-Code, 24-31: Umschalttasten.
```

~GEMDOS(2,z%) **cconout()**

Zeichen auf dem Bildschirm ausgeben (vgl. OUT 2,z%).

```
z%  Bit 0-7: ASCII-Code des Zeichens.
```

r%=GEMDOS(3) **cauxin()**

Liest ein Zeichen von der seriellen Schnittstelle (vgl. INP(1)).

```
r%  Code des gelesenen Zeichens.
```

~GEMDOS(4,z%) **cauxout()**

Zeichen auf serieller Schnittstelle ausgeben (vgl. OUT 1,z%).

```
z%  Code des auszugebenden Zeichens.
```

~GEMDOS(5,z%) **cprnout()**

Zeichen an Drucker-Schnittstelle ausgeben (vgl. OUT 0,z%).

z% Code des auszugebenden Zeichens.

r%=GEMDOS(6,z%) **crawio()**

Zeichen von Tastatur ohne Bildschirmanzeige lesen oder schreiben (vgl. OUT 2,z% sowie INKEY$).

r% bei z%=255 das von der Tastatur gelesene Zeichen.
z% Code des auszugebenden Zeichen, bei z%=255 Zeichen einlesen.

r%=GEMDOS(7) **crawcin()**

Zeichen von Tastatur ohne Bildschirmanzeige lesen (vgl. INP(2)).

r% Code des gelesenen Zeichens.

r%=GEMDOS(8) **cnecin()**

Wie GEMDOS(7), aber ohne Beachtung von Steuerzeichen, z.B. CTRL+C.

r% Code des gelesenen Zeichens.

~GEMDOS(9,L:adr%) **cconws()**

Gibt eine Zeichenkette auf dem Bildschirm aus.

adr% Adresse des Strings, der mit einem Null-Byte enden muß.

~GEMDOS(10,L:adr%) **cconrs()**

Liest eine Zeichenkette von der Tastatur ein (CTRL+C führt zu Absturz).

adr% Puffer für String: erstes Byte ist Anzahl zu lesender Zeichen, zweites Byte Anzahl gelesener Zeichen, dann folgt String.

rl=GEMDOS(11) **cconis()**

Prüft, ob ein Zeichen im Tastaturpuffer ist.

rl Ist TRUE, wenn ein Zeichen anliegt, sonst FALSE.

~GEMDOS(14,d%) **dsetdrv()**

Legt aktuelles Laufwerk fest; 0=A, 1=B, usw. (vgl. CHDRIVE).

d% Nummer des Laufwerks.

rl=GEMDOS(16) **cconos()**

Prüft, ob ein Zeichen auf dem Bildschirm ausgeben werden kann.

rl Ausgabestatus, ist immer TRUE.

rl=GEMDOS(17) **cprnos()**

Prüft, ob auf der Parallel-Schnittstelle ausgegeben werden kann.

rl Ausgabestatus, TRUE wenn empfangsbereit (normalerweise Drucker).

r! = GEMDOS(18) **cauxis()**

Prüft, ob an der seriellen Schnittstelle ein Zeichen anliegt.

```
r!  TRUE wenn Zeichen anliegt, sonst FALSE.
```

r! = GEMDOS(19) **cauxos()**

Prüft Ausgabestatus der seriellen Schnittstelle.

```
r!  TRUE wenn ein Zeichen ausgegeben werden kann, sonst FALSE.
```

r% = GEMDOS(25) **dgetdrv()**

Ermittelt aktuelles Laufwerk.

```
r%  Nummer des aktuellen Laufwerks (A=0, B=1, usw.).
```

~ GEMDOS(26,L:adr%) **fsetdta()**

Setzt Disk-Transfer-Adresse (DTA), normalerweise BASEPAGE + 128.

```
adr%  Die zu setzende Adresse.
```

r% = GEMDOS(42) **tgetdate()**

Ermittelt das Systemdatum (vgl. DATE$).

```
r%  Datum: Bit 0-4: Tag, 5-8: Monat, 9-15: Jahr minus 1980.
```

~ GEMDOS(43,d%) **tsetdate()**

Setzt das Systemdatum (vgl. SETTIME).

```
d%  Das neue Datum (Format siehe GEMDOS(42)).
```

r% = GEMDOS(44) **tgettime()**

Ermittelt die Systemuhrzeit (vgl. TIME$).

```
r%  Uhrzeit: Bit 0-4: Sekunden, 5-10: Minuten, 11-15: Stunden.
```

~ GEMDOS(45,t%) **tsettime()**

Setzt die Systemuhrzeit (vgl. SETTIME).

```
t%  Die neue Uhrzeit (Format siehe GEMDOS(44)).
```

r% = GEMDOS(47) **fgetdta()**

Ermittelt die aktuelle Disk-Transfer-Adresse (DTA).

```
r%  Die ermittelte Adresse.
```

r% = GEMDOS(48) **sversion()**

Ermittelt die Nummer der GEMDOS-Version.

```
r%  Die Versionsnummer.
```

~ GEMDOS(49,L:b%,r%) **ptermres()**

Beendet das Programm und hält es resident (in GFA-BASIC nicht verwendbar).

```
b%  Anzahl resident zu haltender Bytes ab BASEPAGE.
r%  Übergabewert an aufrufendes Programm.
```

r% = GEMDOS(54,L:adr%,d%) **dfree()**

Ermittelt Daten über den Speicherplatz einer Diskette (vgl. DFREE).

```
r%    -46 falls eine falsche Laufwerksnummer angegeben wurde.
adr%  Adresse einer 4 mal 4 Byte großen Disk-Info-Struktur:
      Long 1: Anzahl freier Cluster, Long 2: Gesamtzahl der Cluster,
      Long 3: Bytes pro Sektor,      Long 4: Sektoren pro Cluster.
```

r% = GEMDOS(57,L:adr%) **dcreate()**

Legt einen Ordner im aktuellen Verzeichnis an (vgl. MKDIR).

```
r%    -34 oder -36 falls ein Fehler auftrat.
adr%  Adresse des neuen Ordnernamens (muß mit Null-Byte enden).
```

r% = GEMDOS(58,L:adr%) **ddelete()**

Löscht einen Ordner (vgl. RMDIR).

```
r%    -34, -36 oder -65 falls ein Fehler auftrat.
adr%  Adresse des Namens des Ordners (muß mit Null-Byte enden).
```

r% = GEMDOS(59,L:adr%) **dsetpath()**

Wechselt das aktuelle Verzeichnis.

```
r%    -34 falls neues Verzeichnis nicht gefunden.
adr%  Adresse des Verzeichnis-Namens (muß mit Null-Byte enden).
```

r% = GEMDOS(60,L:adr%,a%) **fcreate()**

Legt eine neue Datei an (vgl. OPEN "O").

```
r%    -34, -35 oder -36 falls ein Fehler auftrat.
adr%  Adresse des Dateinamens (muß mit Null-Byte enden)
a%    Bit 0 gesetzt: schreibgeschützte Datei, Bit 1: versteckte Datei,
      Bit 2: Systemdatei (auch versteckt), Bit 3: Diskettenname.
```

r% = GEMDOS(61,L:adr%,m%) **fopen()**

Öffnet eine Datei (vgl. OPEN).

```
r%    -33, -35 oder -36 im Fehlerfall, sonst Datei-Handle.
adr%  Adresse des Dateinamens (muß mit Null-Byte enden).
m%    0 für Lesen, 1 für Schreiben, 2 für Lesen und Schreiben.
```

r% = GEMDOS(62,h%) **fclose()**

Schließt eine Datei (vgl. CLOSE).

```
r%  -37 falls Fehler aufgetreten.
h%  Datei-Handle der zu schließenden Datei.
```

r% = GEMDOS(63,h%,L:l%,L:adr%) **fread()**

Liest Bytes aus einer offenen Datei (vgl. BGET).

```
r%    -37 im Fehlerfall, sonst Anzahl gelesener Bytes.
h%    Datei-Handle.
l%    Anzahl zu lesender Bytes.
adr%  Adresse, an die die Bytes geschrieben werden sollen.
```

r% = GEMDOS(64,h%,L:l%,L:adr%) **fwrite()**

Schreibt Bytes in eine Datei (vgl. BPUT).

```
r%    -36 oder -37 im Fehlerfall, sonst Zahl der geschriebenen Bytes.
h%    Datei-Handle.
l%    Anzahl der zu schreibenden Bytes.
adr%  Adresse, ab der die zu schreibenden Bytes im Speicher stehen.
```

r% = GEMDOS(65,L:adr%) **fdelete()**

Löscht eine Datei (vgl. KILL).

```
r%    -33 oder -36 falls ein Fehler auftrat.
adr%  Adresse des Namens der Datei (muß mit Null-Byte enden).
```

r% = GEMDOS(66,L:n%,h%,m%) **fseek()**

Setzt den Zeiger für Dateizugriffe neu (vgl. SEEK, RELSEEK).

```
r%  -32 oder -37 falls ein Fehler auftrat.
n%  Anzahl zu überspringender Bytes.
m%  0: ab Dateianfang, 1: ab aktueller Position, 2: ab Dateiende.
```

r% = GEMDOS(67,L:adr%,m%,a%) **fattrib()**

Liest oder schreibt Dateiattribute.

```
r%    -33 oder -34 im Fehlerfall, sonst bisherige Dateiattribute.
adr%  Adresse des Dateinamens (muß mit Null-Byte enden).
m%    0: Dateiattrubute lesen, 1: Dateiattribute schreiben.
a%    Dateiattribute: Bit 0: Schreibgeschützt, 1: Versteckt,
      2: Systemdatei, 3: Diskettenname, 4: Ordner, 5: Archiv-Bit.
```

r%=GEMDOS(69,h%) **fdup()**

Erzeugt ein zweites Datei-Handle.

```
r%  -35 oder -37 falls ein Fehler auftrat, sonst zweites Dateihandle.
h%  Das Original-Datei-Handle.
```

r%=GEMDOS(70,h%,nh%) **fforce()**

Legt ein neues Ausgabe-Handle für GEMDOS-Ausgaben fest.

```
r%   -37 falls ein Fehler auftrat.
h%   Handle des umzuleitenden Datenkanals.
nh%  Handle des Kanals, in den die Ausgaben umgeleitet werden.
```

r%=GEMDOS(71,L:adr%,d%) **dgetpath()**

Ermittelt den aktuellen Zugriffspfad eines Laufswerks (vgl. DIR$).

```
r%    -46 falls ein Fehler auftrat.
adr%  Adresse, ab der der Zugriffspfad abgelegt wird.
d%    Laufswerkskennung (0=aktuelles, 1=A, 2=B, usw.).
```

r%=GEMDOS(72,L:b%) **malloc()**

Speicherplatz reservieren oder ermitteln (vgl. MALLOC).

```
r%  für b%=-1 Länge des größten freien Speicherbereichs, sonst
    Anfangsadresse des reservierten Bereichs oder Fehlermeldung.
b%  Anzahl zu reservierender Bytes, für b%=-1 siehe r%.
```

r% = GEMDOS(73,L:adr%) **mfree()**

Gibt einen mit GEMDOS(72) reservierten Bereich frei (vgl. MFREE).

```
r%    -40 falls Fehler auftrat.
adr%  Adresse des freizugebenden Speicherbereichs.
```

r% = GEMDOS(74,L:adr%,L:b%) **mshrink()**

Verkürzt einen mit GEMDOS(72) reservierten Bereich (vgl. MSHRINK).

```
r%    -40 oder -67 falls ein Fehler auftrat.
adr%  Adresse des zu verkleinernden Speicherbereichs.
b%    Neue Länge des Speicherbereichs in Bytes.
```

r% = GEMDOS(75,m%,L:p%,L:c%,L:e%) **pexec()**

Führt ein Programm als Unterprogramm aus (vgl. EXEC).

```
r%  -32, -33, -39 oder -66 im Fehlerfall.
m%  0: Laden und Starten, 3: Laden, 4: Starten, 5: Basepage erzeugen.
p%  Adresse des Programmnamens oder bei m%=4 der Basepage.
c%  Adresse der Kommandozeile (nicht bei m%=4)
e%  Adresse des Environment-Strings (nicht bei m%=4)
```

~ GEMDOS(76,r%) **pterm()**

Beendet das laufende Programm (vgl. QUIT, SYSTEM).

```
r%  Rückgabewert an das aufrufende Programm.
```

r%=GEMDOS(78,L:adr%,a%) **fsfirst()**

Sucht im aktuellen Inhaltsverzeichnissen nach Dateien mit bestimmten Namen. Der gefundene Dateiname wird in der DTA abgelegt.

```
r%    -33: Datei nicht gefunden, -49: keine weiteren Dateien.
adr%  Adresse des Dateinamens (darf die Wildcards * und ? enthalten).
a%    Dateiattribute: Bit 0: Schreibgeschützt, 1: Versteckt,
      2: Systemdatei, 3: Diskettenname, 4: Ordner, 5: Archiv-Bit.
```

r%=GEMDOS(79) **fsnext()**

Setzt die mit GEMDOS(78) begonnene Suche fort.

```
r%  -49 falls keine weiteren Dateien mehr, auf die der Such-String paßt.
```

r%=GEMDOS(86,0,L:o%,L:n%) **frename()**

Benennt eine Datei um (vgl. NAME, RENAME).

```
r%  -34 oder -36 falls ein Fehler auftrat.
o%  Adresse des alten Dateinamens.
n%  Adresse des neuen Dateinamens.
```

~GEMDOS(87,L:adr%,h%,m%) **fdatime()**

Setzt oder ermittelt Uhrzeit und Datum einer Datei (vgl. TOUCH).

```
adr%  Adresse der Zeit-Information (4 Byte).
h%    Datei-Handle.
m%    0: Lesen der Dateizeit, 1: Setzen der Dateizeit.
```

BIOS-Tabelle

~BIOS(0,L:adr%) getmpb()

Initialisierung des Memory-Parameter-Block.

```
adr%  Adresse des neuen MPB.
```

r%=BIOS(1,d%) constat()

Stellt den Eingabestatus einer Einheit fest (vgl. INP?).

```
r%  0: kein zeichen verfügbar, -1: Zeichen verfügbar.
d%  0: Parallel-Schnittstelle, 1: Serielle Schnittstelle,
    2: Tastatur, 3: MIDI-Schnittstelle.
```

r%=BIOS(2,d%) bconin()

Liest ein Zeichen von einer Einheit ein (vgl. INP).

```
r%  Eingelesenes zeichen (8 Bit).
d%  0: parallel-Schnittstelle, 1: Serielle Schnittstelle,
    2: Tastatur, 3: MIDI-Schnittstelle.
```

~BIOS(3,d%,b%) bconout()

Gibt ein Zeichen auf Einheit aus (vgl. OUT).

```
d%  0: Parallel-Schnittstelle, 1: Serielle Schnittstelle,
    2: Tastatur, 3: MIDI-Schnittstelle, 4: IKBD.
b%  Das auszugebende Zeichen.
```

r%=BIOS(4,f%,L:b%,n%,s%,d%) **rwabs()**

Lesen und Schreiben von Sektoren auf Diskette.

```
r%  0 falls kein Fehler auftrat.
f%  0: Lesen, 1: Schreiben.
b%  Adresse des Daten-Speicherbereichs.
n%  Anzahl Sektoren.
s%  Nummer des Startsektors.
d%  Nummer des Laufwerks (0=A, 1=B, usw.).
```

r%=BIOS(5,n%,L:adr%) **setexec()**

Setzen und Lesen der Exception-Vektoren.

```
r%    Für adr%=-1 der bisherige Wert des Vektors.
n%    Nummer des Exception-Vektors.
adr%  Neue Adresse des Vektors oder -1 (siehe r%).
```

r%=BIOS(6) **tickcal()**

Abfrage des System-Timers.

```
r%  Anzahl der verstrichenen Millisekunden (in 20ms Auflösung).
```

BIOS(7,d%) **getbpb()**

Ermittelt die Adresse des Bios-Parameter-Blocks eines Laufwerks.

```
r%  Adresse des BPB.
d%  Nummer des Laufwerks (0=A, 1=B, usw.).
```

r! = BIOS(8,d%) **bcostat()**

Ermittelt Ausgabestatus einer Einheit (vgl. OUT?).

```
r!  TRUE falls zeichen gesendet werden kann, sonst FALSE.
d%  0: Parallel-Schnittstelle, 1: Serielle Schnittstelle,
    3: MIDI-Schnittstelle, 4: Tastatur-Chip (IKBD).
```

r% = BIOS(9,d%) **mediach()**

Prüft, ob Diskette gewechselt wurde.

```
r%  0: bestimmt nicht, 1: vielleicht, 2: gewechselt.
d%  Nummer des Laufwerks (0=A, 1=B, usw.).
```

r% = BIOS(10) **drvmap()**

Prüft, welche Laufwerke angeschlossen sind.

```
r%  Bit-Vektor, gesetztes Bit pro Laufwerk (Bit 0=A, Bit 1=B, usw.).
```

r% = BIOS(11,c%) **kbshift()**

Ermittelt oder setzt den Status der Tastaturumschalttasten.

```
r%  Für c%=-1 aktueller Status der Umschalttasten.
c%  Neuer Status, Bit 0: Shift links, Bit 1: Shift rechts,
    Bit 2: Control, Bit 3: Alternate, Bit 4: Caps-Lock,
    Bit 5: Alternate+Clr/Home (rechte Maustaste),
    Bit 6: Alternate+Insert (linke Maustaste).
```

XBIOS-Tabelle

~XBIOS(0,t%,L:p%,L:v%) **initmous()**

Initialisiert Mausbehandlungs-Routinen (nicht mit GEM verträglich).

```
t%   0: Maus ausschalten, 1: Maus in relativen Modus schalten,
     2: Maus in absoluten Modus schalten, 4: Maus in Tastaturmodus.
p%   Adresse einer Informationsstruktur.
v%   Adresse der Mausbehandlungsroutine.
```

r%=XBIOS(2) **physbase()**

Ermittelt die Adresse des physikalischen Bildschirmspeichers.

```
r%  Adresse des physikalischen Bildschirmspeichers.
```

r%=XBIOS(3) **logbase()**

Ermittelt die Adresse des logischen Bildschirmspeichers.

```
r%  Adresse des logischen Bildschirmspeichers.
```

r%=XBIOS(4) **getrez()**

Ermittelt die aktuelle Bildschirmauflösung.

```
r%  0: 320x200, 1: 640*200, 2: 640x400, 3: reserviert.
```

~XBIOS(5,L:l%,L:p%,r%) **setscreen()**

Setzt die Adressen der Bildschirmspeicher und die Auflösung, der Wert -1 bedeutet bei allen Parametern keine Änderung. Adressen nur auf 256-Byte-Grenzen.

```
l%  Neue Adresse des logischen Bildschirmspeichers.
p%  Neue Adresse des physikalischen Bildschirmspeichers.
r%  Neue Bildschirmauflösung (siehe XBIOS(4), nicht mit
    GEM verträglich).
```

~XBIOS(6,L:adr%) **setpalette()**

Lädt eine neue Farbpalette.

```
adr%  Adresse einer Tabelle aus 16 Worten, die die neue Palette enthält.
```

r%=XBIOS(7,n%,c%) **setcolor()**

Setzen oder Abfragen eines Farbregisters (vgl. SETCOLOR).

```
r%  Für c%=-1 Abfrage des angegebenen Farbregisters.
n%  Nummer des Farbregisters (0 bis 15).
c%  Neue Farbe, bei c%=-1 siehe r%.
```

r%=XBIOS(8,L:b%,L:f%,d%,sec%,t%,side%,n%) **floprd()**

Liest Sektoren von der Diskette.

```
r%     0 falls kein Fehler auftrat.
b%     Adresse des Bereichs, in den die Sektoren gelesen werden.
f%     Unbenutzt.
d%     Nummer des Laufwerks (0=A, 1=B, usw.).
sec%   Nummer des Sektors, ab dem gelesen wird.
t%     Nummer der Spur (Track), von der gelesen wird.
side%  Seite des Diskette (0 oder 1).
n%     Anzahl zu lesender Sektoren (müssen auf einem Track liegen).
```

r%=XBIOS(9,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopwr()**

Schreibt Sektoren auf Diskette.

```
r%      0 falls kein Fehler auftrat.
b%      Adresse des Speicherbereichs mit den zu schreibenden Daten.
f%      Unbenutzt.
d%      Nummer des Laufwerks (0=A, 1=B, usw.).
sec%    Nummer des Sektors, ab dem geschrieben wird.
t%      Nummer der Spur (Track), in den geschrieben wird.
side%   Seite der Diskette (0 oder 1).
n%      Anzahl der zu schreibenden Sektoren (alle auf einem Track).
```

r%=XBIOS(10,L:b%,L:f%,d%,sec%,t%,side%,i%,L:m%,v%)flopfmt()

Formatiert eine Spur der Diskette.

```
r%      0 falls kein Fehler auftrat.
b%      Adresse eines Bereichs für Zwischenspeicherungen (min. 8KB).
f%      Unbenutzt.
d%      Nummer des Laufwerks (0=A, 1=B,usw.).
sec%    Sektoren pro Track (normalerweise 9).
t%      Nummer der zu formatierenden Spur (Track).
side%   Seite der Diskette (0 oder 1).
i%      Interleave-Faktor (normalerweise 1).
m%      Magic-number &H87654321.
v%      Wert in Sektoren nach Formatierung (normalerweise &HE5E5).
```

~XBIOS(12,n%,L:adr%) **midiws()**

Gibt einen Speicherbereich auf der MIDI-Schnittstelle aus.

```
n%    Anzahl der auszugebenden Bytes minus 1.
adr%  Adresse des auszugebenden Speicherbereichs.
```

~XBIOS(13,n%,L:adr%) **mfpint()**

Interrupt-Vektoren des MFP ändern (nicht verwendbar in GFA-BASIC).

```
n%    Nummer des Interrupts (0 bis 15).
adr%  Neue Adresse des einzustellenden Interrupts.
```

r%=XBIOS(14,d%) **iorec()**

Ermittelt Adresse von Ein-/Ausgabe-Einheiten.

```
r%  Adresse des Datenpuffers für die Ein- und Ausgabe der Einheit.
d%  0: RS 232, 1: IKBD, 2: MIDI.
```

~XBIOS(15,b%,h%,ucr%,rsr%,tsr%,scr%) **rsconf()**

Setzt die Parameter für die serielle Schnittstelle.

```
b%    Baudrate.
h%    Handshake-Modus, 0: ohne, 1: XON/XOFF, 2: RTS/CTS, 3: beides.
ucr%  USART-Kontrollregister des MFP.
rsr%  Receiver-Statusregister des MFP.
tsr%  Transmitter-Statusregister des MFP.
scr%  Synchronus-Character-Register des MFP.
```

r%=XBIOS(16,L:us%,L:sh%,L:cl%) **keytbl()**

Änderung der Adressen der Tastaturbelegungstabellen.

```
r%   Adresse der KEYTAB-Struktur.
us%  Adresse der Tabelle für Tasten ohne Shift.
sh%  Adresse der Tabelle für Tasten mit Shift.
cl%  Adresse der Tabelle mit Caps-Lock.
```

r%=XBIOS(17) **random()**

Ermittelt eine Zufallszahl (vgl. RAND, RANDOM).

```
r%  Eine 24-Bit-Zufallszahl (0 bis 16777215).
```

~XBIOS(18,L:b%,L:s%,d%,f%) **protobt()**

Erzeugt einen Boot-Sektor auf der Diskette.

```
b%  Adresse eines Puffers (512 Byte) zur Erzeugung des Bootsektors.
s%  Seriennummer, -1: nehme alte, Bit größer 24 setzen: zufällige.
d%  Diskettentyp (Tracks/Seiten), 0: 40/1, 1: 40/2, 2: 80/2, 3: 80/2).
f%  0: nicht ausführbarer Bootsektor, 1: ausführbar, -1: nicht ändern.
```

r%=XBIOS(19,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopver()**

Vergleicht Sektoren der Diskette mit Speicherbereichinhalt.

```
b%     Adresse des Seicherbereichs, mit dem verglichen wird.
f%     Unbenutzt.
d%     Nummer des Diskettenlaufwerks (0=A, 1=B, usw.).
sec%   Nummer des Sektors, ab dem verglichen wird.
t%     Nummer der Spur (Tracks).
side%  Seite der Diskette (0 oder 1).
n%     Anzahl der zu vergleichenden Sektoren.
```

~XBIOS(20) **scrdmp()**

Ruft die Hardcopy-Routine auf (vgl. HARDCOPY).

r%=XBIOS(21,c%,s%) **curscon()**

Legt Cursor-Attribute fest.

```
r%  Falls c%=5 ergibt dies die Cursor-Blinkfrequenz.
c%  0: Cursor ausschalten, 1: Cursor einschalten,
    2: Cursor-Blinken einschalten, 3: Blinken ausschalten,
    4: Blinkrate auf Wert in s% einstellen, 5: siehe r%.
s%  Falls c%=4, Blinkrate auf s% setzen.
```

~XBIOS(22,L:t%) **bsettime()**

Stellt Datum und Zeit ein (vgl. SETTIME).

```
t%  Bit 0-4: Sekunden, 5-10: Minuten, 11-15: Stunden, 16-20: Tag,
    21-24: Monat, 25-31: Jahr minus 1980.
```

r%=XBIOS(23) **bgettime()**

Ermittelt Datum und Zeit (vgl. TIME$, DATE$).

```
r%  Bit-Belegung siehe XBIOS(22).
```

~XBIOS(24) **bioskey()**

Stellt die Originaltastaturbelegung ein (vgl. XBIOS(16)).

XBIOS(25,n%,L:adr%) **ikbdws()**

Sendet Daten an den Tastatur-Chip (IKBD).

```
n%    Anzahl zu sendender Bytes minus 1.
adr%  Adresse an der die zu sendenden Daten stehen.
```

~XBIOS(26,i%) **jdisint()**

Sperrt einen MFP-Interrupt.

i% Nummer des zu sperrenden Interrupts (0-15).

~XBIOS(27,i%) **jenabint()**

Gibt einen Interrupt des MFP frei.

i% Nummer des Interrupts.

r%=XBIOS(28,d%,reg%) **giaccess()**

Liest und schreibt Register des Sound-Chips.

r% Falls Lesezugriff, steht hier der Registerwert.
d% Falls Schreibzugriff, ist dies der zu schreibende Wert (8 Bit).
r% Nummer des Registers (0 bis 15), bei gesetztem Bit 7 Schreibzugriff.

~XBIOS(29,b%) **offgibit()**

Löscht ein Bit des PORT-A-Registers des Sound-Chips.

b% Bit-Muster, das mit dem vorhandenen OR-verknüpft wird.

~XBIOS(30,b%) **ongibit()**

Setzt ein Bit des PORT-A-Registers des Sound-Chips.

b% Bit-Muster, das mit dem vorhandenen AND-verknüpft wird.

XBIOS(31,t%,c%,d%,L:adr%) **xbtimer()**

Setzt die MFP-Timer.

```
t%    Nummer des Timers (0 bis 3).
c%    Control-Register.
d%    Data-Register.
adr%  Adresse der Timer-Interrupt-Routine.
```

~XBIOS(32,L:adr%) **dosound()**

Startet eine Sound-Sequenz, die im Interrupt abgearbeitet wird.

```
adr%  Adresse des Speicherbereichs der Sound-Befehle.
```

r%=XBIOS(33,c%) **setprt()**

Setzt oder liest die Druckerparameter.

```
r%  aktuelle Konfiguration, falls c%=-1.
c%  Bit nicht gesetzt/gesetzt, 0: Matrixdrucker/Typenrad, 1: Farbe/
    Monochrom, 2: 1280/960 Punkte pro Zeile, 3: Draft/NLQ,
    4: parallel/seriell, 5: Endlospapier/Einzelblatt.
```

r%=XBIOS(34) **kbdvbas()**

Adresse einer Tabelle mit Vektoren des Tastaturprozessors.

```
r%  ermittelte Adresse.
```

r%=XBIOS(35,a%,w%) **kbrate()**

Setzt oder liest Tastatur-Wiederholrate.

```
r%  aktuelle Daten, Bit 0-7: Wiederholrate, 8-15: Ansprechzeit.
a%  Ansprechverzögerung.
w%  Wiederholrate.
```

~XBIOS(36,L:adr%) **prtblk()**

Hardcopy-Routine mit Prameterblock.

```
adr%  Adresse eines Parameterblocks für die Hardcopy-Routine.
```

~XBIOS(37) **vsync()**

Wartet auf den nächsten Vertical-Blank-Interrupt (vgl. VSYNC).

~XBIOS(38,L:adr%) **supexec()**

Aufruf einer Assembler-Routine im Supervisormodus (ohne Systemaufrufe).

```
adr%  Adresse der Assembler-Routine.
```

~XBIOS(39) **puntaes()**

Schaltet die AES ab, sofern sie sich nicht im ROM befinden.

r%=XBIOS(64,b%) **blitmode()**

Ansteuerung und Abfrage des Blitters (nur im Blitter-TOS).

```
r%  Aktueller Blitter-Status, wenn b%=-1 ist, Bit 1: Blitter da?.
b%  -1: siehe r%, sonst Bit 0: gesetzt Blitter an, sonst Blitter aus,
    Bit 1-14: reserviert (-1), Bit 15: 0.
```

Tabelle der Line-A-Variablen

Mit L~A erhält man die Basisadresse der Line-A-Variablen:

{L~A-906}	Adresse des aktuellen Font-Headers.
L~A-856 ..	37 Worte, aktives DEFMOUSE.
{L~A-460}	Zeiger auf den aktuellen Font-Header.
L~A-456	Array aus vier Zeigern, von denen der letzte Null sein muß. Jeder Zeiger zeigt auf eine verkettete Liste von Zeichensätzen. Die ersten beiden Zeiger gelten für residente Fonts. Der dritte ist der für die GDOS-Fonts, dieser wird bei jedem VDI-Aufruf zurückgesetzt.
INT{L~A-440}	Gesamtzahl dieser Fonts.
INT{L~A-46}	Textzeilenhöhe.
INT{L~A-44}	Maximale Cursor-Spalte.
INT{L~A-42}	Maximale Cursor-Zeile.
INT{L~A-40}	Länge einer Textzeile in Byte.
INT{L~A-38}	Texthintergrundfarbe.
INT{L~A-36}	Textvordergrundfarbe.
{L~A-34}	Adresse des Cursors im Bildschirmspeicher.
INT{L~A-30}	Abstand der ersten Textzeile vom oberen Bildschirmrand.
INT{L~A-28}	CRSCOL.
INT{L~A-26}	CRSLIN.
BYTE{L~A-24}	Cursor-Blinkperiode.
BYTE{L~A-23}	Cursor-Blinkzähler
{L~A-22}	Adresse der Font-Daten für den Monochrom-Modus.
INT{L~A-18}	Letztes ASCII-Zeichen des Fonts.
INT{L~A-16}	Erstes ASCII-Zeichen des Fonts.
INT{L~A-12}	Horizontale Auflösung in Pixel.
{L~A-10}	Adresse der Font-Offset-Tabelle
INT{L~A-4}	Vertikale Auflösung in Pixel.
INT{L~A-0}	Anzahl der Bitplanes.
INT{L~A+2}	Bytes pro Bildschirmzeile.
{L~A+4}	Zeiger auf das CONTRL-Feld.
{L~A+8}	Zeiger auf das INTIN-Feld.
{L~A+12}	Zeiger auf das PTSIN-Feld.
{L~A+16}	Zeiger auf das INTOUT-Feld.

{L~A+20}	Zeiger auf das PTSOUT-Feld.
INT{L~A+24}	Farbwert für Bit-Ebene 0.
INT{L~A+26}	Farbwert für Bit-Ebene 1.
INT{L~A+28}	Farbwert für Bit-Ebene 2.
INT{L~A+30}	Farbwert für Bit-Ebene 3.
INT{L~A+32}	Flag, letztes Pixel einer Linie nicht zeichnen
INT{L~A+34}	Linienmuster.
INT{L~A+36}	Graphmode
INT{L~A+38} bis	
INT{L~A+44}	2 Koordinaten-Paare
{L~A+46}	Zeiger auf aktuelles Füllmuster.
{L~A+50}	Zeiger auf die aktuelle Füllmustermaske.
INT{L~A+52}	Flag für vielfarbiges Füllen.
INT{L~A+54}	Clipping-Flag.
INT{L~A+56} bis	
INT{L~A+64}	Clipping-Koordinaten.
INT{L~A+66}	Vergrößerungsfaktor.
INT{L~A+68}	Vergrößerungsrichtung.
INT{L~A+70}	Flag für Proportionalschrift.
INT{L~A+72}	x-Offset für Textblt.
INT{L~A+74}	y-Offset für Textblt.
INT{L~A+76}	x-Koordinate eines Zeichens auf dem Bildschirm.
INT{L~A+78}	y-Koordinate eines Zeichens auf dem Bildschirm.
INT{L~A+80}	Breite eines Zeichens.
INT{L~A+82}	Höhe eines Zeichens.
{L~A+84}	Zeiger auf Zeichensatz-Image.
{L~A+88}	Breite des Zeichensatz-Image.
INT{L~A+90}	Textstil.
INT{L~A+92}	Maske für schattierte Textausgabe.
INT{L~A+94}	Maske für Kursivschrift.
INT{L~A+96}	Zusätzliche Breite für Fettschrift.
INT{L~A+98}	Kursivschrift-Offset rechts.
INT{L~A+100}	Kursivschrift-Offset links.
INT{L~A+102}	Vergrößerungs-Flag.
INT{L~A+104}	Textrotationswinkel.

INT{L~A+106}	Textfarbe.
{L~A+108}	Zeiger auf Puffer für Texteffekte.
INT{L~A+112}	Offset für einen zweiten Puffer dieser Art.
INT{L~A+114}	Farbe des Texthintergrundes.
INT{L~A+116}	Flag für copy raster form, <>0 für transparent.
{L~A+118}	Zeiger auf eine Routine, die den Abbruch eines Füllvorgangs ermöglicht, bei 3.0 durch Shift-Alternate-Control verändert.

Tabelle der Eingabe-Parameter für V_OPN(v)WK

Außer x optional, dann folgende Default-Werte:

x Geräteidentifikationsnummer (Standard)

```
 1 ..: Bildschirm        11 ..: Plotter
 21 ..: Drucker          31 ..: Metafile
 41 ..: Kamera           51 ..: Grafiktablett

1  Linientyp.
1  Linienfarbe.
1  Markierungstyp.
1  Markierungsfarbe.
1  Textstil.
1  Textfarbe.
1  Fülltyp.
1  Füllstill.
1  Füllfarbe.
2  Koordinatensystem (0: NDC, 1: reserviert, 2: RC)
```

Tabelle des WORK_OUT-Feldes des VDI

Bei V_OPN(v)WK sind die Ergebniswerte in INTOUT(0) bis INTOUT(44) und in PTSOUT(0) bis PTSOUT(11).

WORK_OUT(0)	Maximale Bildbreite in Pixel.
WORK_OUT(1)	Maximale Bildhöhe in Pixel.
WORK_OUT(2)	0: Exakte Bildskalierung möglich, 1: Nicht möglich.
WORK_OUT(3)	Breite eines Pixel in Mikrometer.
WORK_OUT(4)	Höhe eines Pixel in Mikrometer.
WORK_OUT(5)	Anzahl der Schriftzeichenhöhen (0: Änderbar).
WORK_OUT(6)	Anzahl der Linientypen.
WORK_OUT(7)	Anzahl der Linienbreiten (0: Änderbar).

WORK_OUT(8) Anzahl der Markierungssymbole.
WORK_OUT(9) Anzahl der Markierungssymbolgrößen (0: Änderbar).
WORK_OUT(10) Anzahl der Zeichensätze.
WORK_OUT(11) Anzahl der Muster.
WORK_OUT(12) Anzahl der Schraffierungsmuster.
WORK_OUT(13) Anzahl vordefinierter Farben.
WORK_OUT(14) Anzahl der Grafikgrundfunktionen (GDP).
WORK_OUT(15) bis
WORK_OUT(24) Liste der Grafikgrundfunktionen (GDP). Zehn Grundfunktionen werden unterstützt:

```
 1: bar        2: arc        3: pie
 4: circle     5: ellipse    6: elliptical arc
 7: elliptical pie           8: rounded rectangle
 9: filled rounded rectangle
10: justified graphic text
```

Das Ende dieser Liste ist mit -1 markiert.
WORK_OUT(25) bis
WORK_OUT(34) Liste der Attribute der Grafikgrundfunktionen:
0: Linie 1: Marker 2: Text
3: Ausgefüllter Bereich 4: Kein Attribut.
WORK_OUT(35) 0: Farben sind nicht darstellbar, 1: Sind darstellbar.
WORK_OUT(36) 0: Text ist nicht rotierbar, 1: Ist rotierbar.
WORK_OUT(37) 0: Füllfunktionen sind nicht möglich, 1: Möglich.
WORK_OUT(38) 0: CELLARRAY nicht verfügbar, 1: Verfügbar.
WORK_OUT(39) Anzahl darstellbarer Farben: 0: Mehr als 32767, 1: Monochrom, größer 2: Anzahl der Farben.
WORK_OUT(40) 1: Grafik-Cursor-Positionierung nur mit Tastatur, 2: Mit Tastatur und Maus.
WORK_OUT(41) Werteingabegerät: 1: Tastatur, 2: Anderes Gerät.
WORK_OUT(42) Auswahltasten: 1: Funktionstasten, 2: Andere
WORK_OUT(43) Anzahl der String-Eingabegeräte: 1: Tastatur.
WORK_OUT(44) Arbeitsstationstyp: 0: Nur Ausgabe, 1: Nur Eingabe, 2: Ein-/Ausgabe, 3: Reserviert, 4: Metafile.
WORK_OUT(45) Minimale Zeichenbreite.
WORK_OUT(46) Minimale Zeichenhöhe.
WORK_OUT(47) Maximale Zeichenbreite.
WORK_OUT(48) Maximale Zeichenhöhe.

WORK_OUT(49)	Minimale sichtbare Linienbreite.
WORK_OUT(50)	Reserviert, immer 0.
WORK_OUT(51)	Maximale Linienbreite in X-Richtung.
WORK_OUT(52)	Reserviert, immer 0.
WORK_OUT(53)	Minimale Markierungsbreite.
WORK_OUT(54)	Minimale Markierungshöhe.
WORK_OUT(55)	Maximale Markierungsbreite.
WORK_OUT(56)	Maximale Markierungshöhe.

VT 52-Tabelle

Die Routinen des VT 52-Emulators können durch die Ausgabe der Strings der folgenden Tabelle mittels PRINT aufgerufen werden.

```
CHR$(27)+"A";  Cursor rauf (stoppt am oberen Rand).
CHR$(27)+"B";  Cursor runter (stoppt am unteren Rand).
CHR$(27)+"C";  Cursor rechts (stoppt am rechten Rand).
CHR$(27)+"D";  Cursor links (stoppt am linken Rand).
CHR$(27)+"E";  Bildschirm löschen (CLS).
CHR$(27)+"H";  Cursor in linke obere Bildschirmecke (LOCATE 1,1).
CHR$(27)+"I";  Cursor rauf (scrollt am oberen Rand).
CHR$(27)+"J";  Löscht Bildschirm ab Cursor-Position.
CHR$(27)+"K";  Löscht Zeile ab Cursor-Position.
CHR$(27)+"L";  Fügt Leerzeile an der Cursor-Position ein.
CHR$(27)+"M";  Löscht Zeile an Cursor-Position (Rest wird hochgezogen).
CHR$(27)+"Y"+CHR$(s+32)+CHR$(z+32);  Entspricht LOCATE z,s.
CHR$(27)+"b"+CHR$(f);  Wählt f als Schriftfarbe.
CHR$(27)+"c"+CHR$(f);  Wählt f als Hintergrundfarbe.
CHR$(27)+"d";  Löscht Bildschirm bis Cursor-Position.
CHR$(27)+"e";  Cursor einschalten.
CHR$(27)+"f";  Cursor ausschalten.
CHR$(27)+"j";  Cursor-Position speichern.
CHR$(27)+"k";  Cursor an gespeicherte Position von CHR$(27)+"j" setzen.
CHR$(27)+"l";  Zeile löschen, in der der Cursor steht.
CHR$(27)+"o";  Zeile bis Cursor-Position löschen.
CHR$(27)+"p";  Inverse Schrift einschalten.
CHR$(27)+"q";  Inverse Schrift ausschalten.
CHR$(27)+"v";  Automatischen Zeilenüberlauf einschalten.
CHR$(27)+"w";  Automatischen Zeilenüberlauf ausschalten.
```

Scan-Code-Tabelle

54	55	56	57	58	59	5A	5B	5C	5D
3B	3C	3D	3E	3F	40	41	42	43	44

01	78 02	79 03	7A 04	7B 05	7C 06	7D 07	7E 08	7F 09	80 0A	81 0B	82 0C	83 0D	29	0E
0F	10	11	12	13	14	15	16	17	18	19	1A	1B		53
1D	1E	1F	20	21	22	23	24	25	26	27	28		1C	2B
3A	60	2C	2D	2E	2F	30	31	32	33	34	35	36		
	38	39										3A		

62	61	
52	48	47
73 4B	50	74 4D

63	64	65	66
67	68	69	4A
6A	6B	6C	4E
6D	6E	6F	72
70		71	

ASCII-Tabelle

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

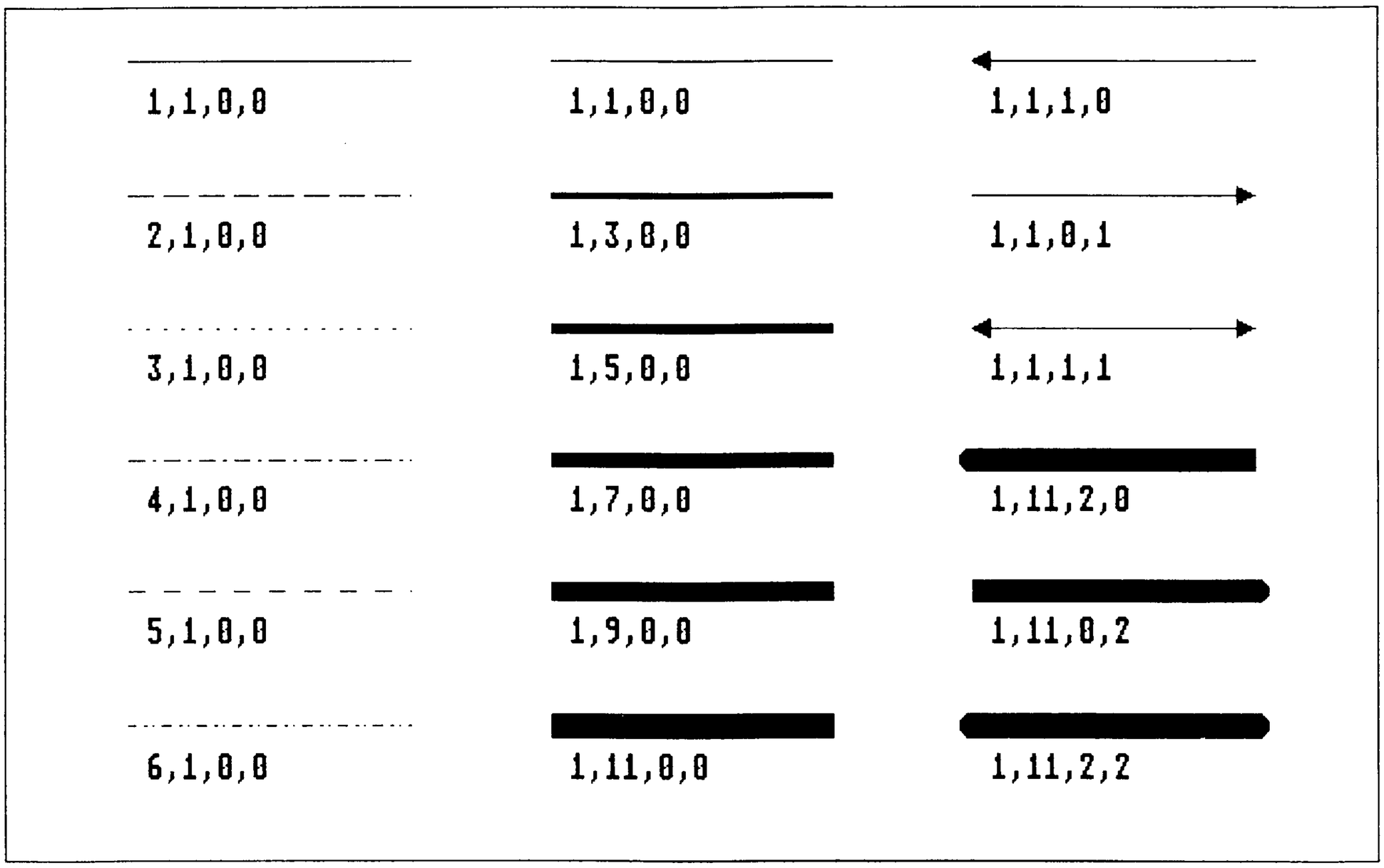
Spezielle ASCII-Zeichen

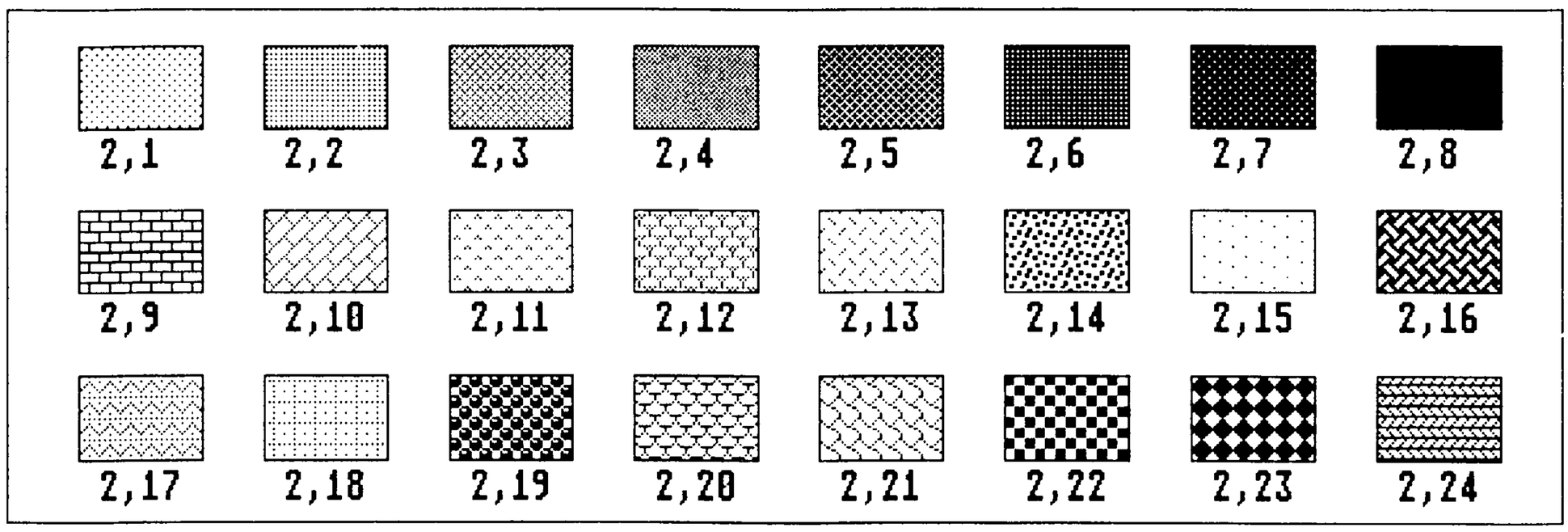
Tabelle der Bezeichnungen der ASCII-Codes 0 bis 31 und 127:

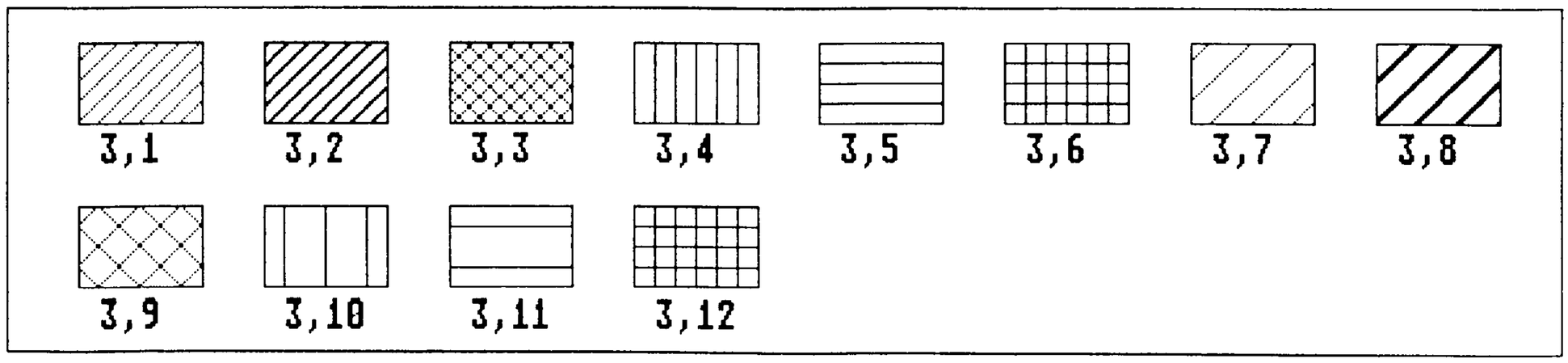
0	NUL	1	SOH	2	STX	3	ETX
4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	LF	11	VT
12	FF	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC
28	FS	29	GS	30	RS	31	US
127	DEL						

ä	Ä	á	à	â	À
132	142	160	133	131	182
ë		é	è	ê	É
137		130	138	136	144
ï		í	ì	î	
139		161	141	140	
ö	Ö	ó	ò	ô	
148	153	162	149	147	
ü	Ü	ú	ù	û	
129	154	163	151	150	

Füllmuster- und Linienstil-Tabelle







Fehlermeldungen

GFA-BASIC-Fehlermeldungen

```
 0 Division durch Null
 1 Überlauf
 2 Zahl nicht Integer|-2147483648 .. 2147483647
 3 Zahl nicht Byte|0 .. 255
 4 Zahl nicht Wort|-32768 .. 32767
 5 Quadratwurzel nur für|positive Zahlen
 6 Logarithmen nur für|Zahlen größer Null
 7 Unbekannter Fehler
 8 Speicher voll
 9 Funktion oder Befehl|noch nicht möglich
10 String zu lang|max. 32767 Zeichen
11 Kein GFA-BASIC-3.00-Programm
12 Programm zu lang|Speicher voll|NEW
13 Kein GFA-BASIC-Programm|EOF - NEW
14 Feld zweimal dimensioniert
15 Feld nicht dimensioniert
16 Feldindex zu groß
17 Dim zu groß
18 Falsche Anzahl Indizes
19 Procedure nicht gefunden
20 Label nicht gefunden
21 Bei Open nur erlaubt:|"I"nput  "O"utput "R"andom|"A"ppend "U"pdate
22 File schon geöffnet
23 File # falsch
24 File nicht geöffnet
25 Falsche Eingabe, keine Zahl
26 Fileende erreicht|EOF
27 Zu viele Punkte für|Polyline/Polyfill|maximal 128
28 Feld muß eindimensional sein
29 Anzahl Punkte größer als Feld
30 Merge - Kein ASCII-File
```

31 Merge - Zeile zu lang - Abbruch
32 ==> Syntax nicht korrekt|Programmabbruch
33 Marke nicht definiert
34 Zu wenig Data
35 Data nicht numerisch
37 Diskette voll
38 Befehl im Direktmodus|nicht möglich
39 Programmfehler|Kein Gosub möglich
40 Clear nicht möglich in|For-Next-Schleifen oder|Proceduren
41 Cont nicht möglich
42 Zu wenig Parameter
43 Ausdruck zu komplex
44 Funktion nicht definiert
45 Zu viele Parameter
46 Parameter falsch|keine Zahl
47 Parameter falsch|kein String
48 Open "R" - Satzlänge falsch
49 Zu viele "R"-Files (max. 31)
50 Kein "R"-File
52 Fields größer als Satzlänge
54 GET/PUT Field-String|Länge falsch
55 GET/PUT Satznummer falsch
60 Sprite-String-Länge falsch
61 Fehler bei RESERVE
62 Menu falsch
63 Reserve falsch
64 Pointer falsch
65 Feldgröße < 256
66 Kein VAR-Array
67 ASIN/ACOS falsch
68 Falsche VAR-Type
69 ENDFUNC ohne RETURN
71 Index zu groß
90 Fehler bei Local
91 Fehler bei For
92 Resume (next) nicht möglich |Fatal, For oder Local
93 Stapel-Fehler

Bomben-Fehlermeldungen

```
100 GFA-BASIC Version 3.00|  Copyright 1986-1988|GFA Systemtechnik GmbH
102 2 Bomben - Bus Error|Peek/Poke falsch?
103 3 Bomben - Adress error|Ungerade Wort-Adresse!|
                            Dpoke/Dpeek, Lpoke/Lpeek?
104 4 Bomben - Illegal Instruction|ungültiger Maschinenbefehl
105 5 Bomben - Divide by Zero|68000 Division durch Null
106 6 Bomben - CHK-Exeption|68000 CHK-Befehl
107 7 Bomben - TRAPV-Exeption|68000 TRAPV-Befehl
108 8 Bomben - Privilege Violation|68000 Privilegverletzung
109 9 Bomben - Trace Exeption|68000 Trace ohne Monitor
```

TOS-Fehlermeldungen

```
 -1 * Allgemeiner Fehler
 -2 * Drive not Ready|Zeitüberschreitung
 -3 * Unbekannter Befehl
 -4 * CRC Fehler|Disk-Prüfsumme falsch
 -5 * Bad Request|Ungültiger Befehl
 -6 * Seek Error|Spur nicht gefunden
 -7 * Unknown Media|Boot-Sector falsch
 -8 * Sektor nicht gefunden
 -9 * Kein Papier
-10 * Schreibfehler
-11 * Lesefehler
-12 * Allgemeiner Fehler 12
-13 * Diskette schreibgeschützt
-14 * Diskette wurde gewechselt
-15 * Unbekanntes Gerät
-16 * Bad Sektor (Verify)
-17 * Andere Diskette einlegen
-32 * Ungültige Funktionsnummer
-33 * Datei nicht gefunden
```

-34 * Pfadname nicht gefunden
-35 * Zu viele Dateien offen
-36 * Zugriff nicht möglich
-37 * Ungültiges Handle
-39 * Speicher voll
-40 * Ungültige Speicherblockadresse
-46 * Ungültige Laufwerksbezeichnung
-49 * Keine weiteren Dateien
-64 * GEMDOS-Bereichsfehler|Seek falsch?
-65 * Interner GEMDOS-Fehler
-66 * Kein Binärprogramm
-67 * Speicherblockfehler

Editor-Fehlermeldungen

Case ohne Select
Select ohne endselect
While ohne Wend
Repeat ohne Until
Do ohne Loop
For ohne next
Wend ohne While
Until ohne Repeat
Loop ohne Do
Next ohne For
If ohne Endif
Endif ohne If
Else ohne If
Else ohne Endif
Exit ohne Schleife
Procedure ohne Return
Procedure in Schleife
Procedure doppelt definiert
Function ohne Endfunc

Function in Schleife
Function doppelt definiert
Return ohne Procedure
Marke doppelt definiert
Local nur in Procedure
Local nicht in Schleife
Function doppelt definiert
Goto in/aus For-Next oder Procedure
Resume in For-Next-Schleife
Resume ohne Procedure
Kein Resume in Function
Endfunc ohne Function

Index

In dieser Liste sind Befehls- und Funktionsnamen aufgeführt. Bei Befehlen ist die Kurzform durch fettere Schrift hervorgehoben. Dahinter stehen Kapitelnummer und Seitenzahl der Beschreibung.

!	7-40
$	1-34
'	7-40
()	3-19
*	2-15, 2-23
*	3-2
+	3-2, 3-12
-	3-2
.	1-22
/	3-2
< =	3-15
< >	3-17
<	3-15
= <	3-15
= = >	1-21
= =	3-14
= >	3-15
=	3-18
> <	3-17
> =	3-15
>	3-15
?	6-7, 6-47
@	7-24
ABS(	4-2
ABSOLUTE	2-24
ACHAR	10-17
ACLIP	10-5
ACOS(	4-9
ADD	4-15
ADD(	4-17
ADDRIN	11-2
ADDRIN(	11-2
ADDRIN() =	11-2
ADDROUT	11-2
ADDROUT(	11-2
ADDROUT() =	11-2
AFTER	7-37
AFTER CONT	7-37
AFTER STOP	7-37
ALERT	9-33
ALINE	10-8
AND(	4-24
AND	3-6
APOLY	10-11
APPL_EXIT(	11-18
APPL_FIND(	11-16
APPL_INIT(	11-13
APPL_READ(	11-14
APPL_TPLAY(	11-17
APPL_TRECORD(	11-17
APPL_WRITE(	11-15
ARECT	10-10
ARRAYFILL	2-6
ARRPTR(	2-23
ASC(	2-8
ASIN(	4-9
ATEXT	10-18
ATN(	4-9
BASEPAGE	2-41
BCHG(	4-20
BCLR(	4-20
BGET	6-42

BIN$(2-10
BIOS(10-1
BITBLT 8-36, 10-12
BLOAD 6-42
BMOVE 2-40
BOUNDARY 8-9
BOX 8-23
BPUT 6-42
BSAVE 6-42
BSET(4-20
BTST(4-20
BYTE(4-27
BYTE{}= 2-19
BYTE{ 2-19

C: 10-39
CALL 10-43
CARD(4-27
CARD{}= 2-19
CARD{ 2-19
CASE 7-8
CFLOAT(2-14
CHAIN 7-48
CHAR{}= 2-19
CHAR{ 2-19
CHDIR 6-30
CHDRIVE 6-30
CHR$(2-8
CINT(2-14
CIRCLE 8-24
CLEAR 2-25
CLEARW 9-26
CLIP 8-16
CLOSE 6-39
CLOSEW 9-23
CLR 2-25
CLS 8-28
COLOR 8-2
CONT 7-8
CONTRL 10-22
CONTRL(10-22
CONTRL()= 10-22
COS(4-9
COSQ(4-9
CRSCOL 6-14
CRSLIN 6-14
CVD(2-12
CVF(2-12
CVI(2-12
CVL(2-12
CVS(2-12

DATA 6-26
DATE$= 2-35
DATE$ 2-35
DEC 4-14
DEFAULT 7-8
DEFBIT 1-30
DEFBYT 1-30
DEFDBL 1-30
DEFFILL 8-6
DEFFLT 1-30
DEFFN 7-29
DEFINT 1-30
DEFLINE 8-10
DEFLIST 1-32
DEFMARK 8-5
DEFMOUSE 8-3
DEFNUM 6-13
DEFSNG 1-30
DEFSTR 1-30
DEFTEXT 8-12
DEFWRD 1-30
DEG(4-9
DELAY 7-42
DELETE 2-32
DFREE(6-30
DIM 2-4
DIM?(2-4
DIR 6-32
DIR$(6-30
DIV 4-15
DIV(4-17
DIV 3-2

DO 7-19
DO UNTIL 7-20
DO WHILE 7-20
DOUBLE{}= 2-19
DOUBLE{ 2-19
DOWNTO 7-14
DPEEK(2-17
DPOKE 2-17
DRAW 8-17, 8-19
DRAW(8-19
DUMP 7-55

EDIT 7-43
ELLIPSE 8-24
ELSE 7-3
ELSE IF 7-5
END 7-43
ENDDO 7-19
ENDFOR 7-14
ENDFUNC 7-27
ENDIF 7-3
ENDPROC 7-24
ENDREPEAT 7-16
ENDSELECT 7-8
ENDSUB 7-24
ENDWHILE 7-18
EOF(6-39
EQV(4-24
EQV 3-11
ERASE 2-25
ERR$(7-35
ERROR 7-35
ERR 7-35
EVEN(4-3
EVERY 7-37
EVERY CONT 7-37
EVERY STOP 7-37
EVNT_BUTTON(11-20
EVNT_DCLICK(11-27
EVNT_KEYBD(11-19
EVNT_MESAG(11-24
EVNT_MOUSE(11-22
EVNT_MULTI(11-26
EVNT_TIMER(11-25
EXEC 10-46
EXEC(10-46
EXIST(6-36
EXIT IF 7-22
EXP(4-8

FALSE 2-34
FATAL 7-35
FGETDTA(6-33
FIELD 6-54
FILES 6-32
FILESELECT 9-35
FILL 8-27
FIX(4-4
FLOAT{}= 2-19
FLOAT{ 2-19
FN 7-29
FOR 7-14
FORM INPUT , 6-6
FORM INPUT AS 6-6
FORM_ALERT(11-44
FORM_BUTTON(11-48
FORM_CENTER(11-46
FORM_DIAL(11-43
FORM_DO(11-42
FORM_ERROR(11-45
FORM_KEYBD(11-47
FRAC(4-4
FRE(2-39
FSEL_INPUT(11-62
FSETDTA(6-33
FSFIRST(6-34
FSNEXT(6-34
FULLW 9-26
FUNCTION 7-27

GB 11-2
GCONTRL 11-2
GCONTRL(11-2
GCONTRL()= 11-2

GDOS? 10-29
GEMDOS(10-1
GEMSYS 11-3
GET 6-55, 8-33
GINTIN 11-2
GINTIN(11-2
GINTIN()= 11-2
GINTOUT 11-2
GINTOUT(11-2
GINTOUT()= 11-2
GOSUB 7-24
GOTO 7-41
GRAF_DRAGBOX(11-51
GRAF_GROWBOX(11-53
GRAF_HANDLE(11-57
GRAF_MKSTATE(11-59
GRAF_MOUSE(11-58
GRAF_MOVEBOX(11-52
GRAF_RUBBERBOX(11-49
GRAF_SHRINKBOX(11-54
GRAF_SLIDEBOX(11-56
GRAF_WATCHBOX(11-55
GRAPHMODE 8-14

HARDCOPY 6-66
HEX$(2-10
HIDEM 6-53
HIMEM 2-41
HLINE 10-9
HTAB 6-14

IF 7-3
IMP(4-23
IMP 3-10
INC 4-14
INFOW 9-26
INKEY$ 6-2
INLINE 2-44, 10-39
INP(# 6-44
INP(6-57
INP?(6-57
INPAUX$ 6-59
INPMID$ 6-59
INPUT 6-3, 6-46
INPUT$(6-45
INSERT 2-32
INSTR(5-6
INT(4-4
INT{}= 2-19
INT{ 2-19
INTIN 10-22
INTIN(10-22
INTIN()= 10-22
INTOUT 10-22
INTOUT(10-22
INTOUT()= 10-22

KEYDEF 6-23
KEYGET 6-18
KEYLOOK 6-18
KEYPAD 6-16
KEYPRESS 6-21
KEYTEST 6-18
KILL 6-41

L: 10-3
LEFT$(5-2
LEN(5-5
LET 2-37
LINE 8-17
LINE INPUT 6-5, 6-46
LIST 7-47
LLIST 7-47
LOAD 7-45
LOC(6-39
LOCAL 7-26
LOCATE 6-7
LOF(6-39
LOG(4-8
LOG10(4-8
LONG{}= 2-19
LONG{ 2-19
LOOP 7-19
LOOP UNTIL 7-20

LOOP WHILE 7-20
LPEEK(2-17
LPOKE 2-17
LPOS(6-66
LPRINT 6-66
LSET 5-10
L ~ A 10-19

MALLOC(2-45
MAX(4-6
MENU 9-16, 9-19
MENU KILL 9-18
MENU OFF 9-18
MENU(9-3
MENU_BAR(11-28
MENU_ICHECK(11-29
MENU_IENABLE(11-30
MENU_REGISTER(11-33
MENU_TEXT(11-32
MENU_TNORMAL(11-31
MFREE(2-45
MID$()= 5-10
MID$(5-3
MIN(4-6
MKD$(2-12
MKDIR 6-35
MKF$(2-12
MKI$(2-12
MKL$(2-12
MKS$(2-12
MOD(4-17
MODE 6-12
MOD 3-2
MONITOR 10-42
MOUSE 6-60
MOUSEK 6-60
MOUSEX 6-60
MOUSEY 6-60
MSHRINK(2-45
MUL 4-15
MUL(4-17

NAME AS 6-41
NEW 7-44
NEXT 7-14
NOT 3-5

OBJC_ADD(11-34
OBJC_CHANGE(11-41
OBJC_DELETE(11-35
OBJC_DRAW(11-36
OBJC_EDIT(11-40
OBJC_FIND(11-37
OBJC_OFFSET(11-38
OBJC_ORDER(11-39
OB_ADR(11-7
OB_FLAGS()= 11-6
OB_FLAGS(11-6
OB_H()= 11-6
OB_H(11-6
OB_HEAD()= 11-4
OB_HEAD(11-4
OB_NEXT()= 11-4
OB_NEXT(11-4
OB_SPEC()= 11-5
OB_SPEC(11-5
OB_STATE()= 11-6
OB_STATE(11-6
OB_TAIL()= 11-4
OB_TAIL(11-4
OB_TYPE()= 11-5
OB_TYPE(11-5
OB_W()= 11-6
OB_W(11-6
OB_X()= 11-6
OB_X(11-6
OB_Y()= 11-6
OB_Y(11-6
OCT$(2-10
ODD(4-3
ON x GOSUB 7-7
ON BREAK CONT 7-32
ON BREAK 7-32
ON BREAK GOSUB 7-32

ON ERROR	7-33
ON ERROR GOSUB	7-33
ON MENU	9-2
ON MENU BUTTON	9-8
ON MENU GOSUB	9-16
ON MENU IBOX	9-11
ON MENU KEY	9-10
ON MENU MESSAGE	9-13
ON MENU OBOX	9-11
OPEN	6-37
OPENW	9-23
OPTION BASE	2-5
OR(	4-24
OR	3-7
OTHERWISE	7-8
OUT	6-44, 6-57
OUT?(	6-57
PAUSE	7-42
PBOX	8-23
PCIRCLE	8-24
PEEK(	2-17
PELLIPSE	8-24
PI	2-34
PLOT	8-17
POINT(	8-26
POKE	2-17
POLYFILL	8-25
POLYLINE	8-25
POLYMARK	8-25
POS(	6-14
PRBOX	8-23
PRED(	4-16, 5-4
PRINT	6-7, 6-47
PROCEDURE	7-24
PSAVE	7-46
PSET	10-6
PTSIN	10-22
PTSIN(	10-22
PTSIN()=	10-22
PTSOUT	10-22
PTSOUT(	10-22
PTSOUT()=	10-22
PTST(	10-7
PUT	6-55, 8-33
QSORT	2-29
QUIT	7-50
RAD(	4-9
RAND(	4-11
RANDOM(	4-11
RANDOMIZE	4-11
RBOX	8-23
RCALL	10-44
RC_COPY	9-32
RC_INTERSECT(	9-30
READ	6-26
RECALL	6-49
RECORD	6-55
RELSEEK	6-51
REM	7-40
RENAME AS	6-41
REPEAT	7-16
RESERVE	2-43
RESTORE	6-26
RESUME	7-33
RETURN	7-24, 7-27
RIGHT$(	5-2
RINSTR(	5-7
RMDIR	6-35
RND(	4-11
RND	4-11
ROL&(	4-22
ROL(	4-22
ROL|(	4-22
ROR&(	4-22
ROR(	4-22
ROR|(	4-22
ROUND(	4-5
RSET	5-10
RSRC_FREE(	11-76
RSRC_GADDR(	11-77
RSRC_LOAD(	11-75

RSRC_OBFIX(11-79
RSRC_SADDR(11-78
RUN 7-49

SAVE 7-46
SCRP_READ(11-60
SCRP_WRITE(11-61
SDPOKE 2-17
SEEK 6-51
SELECT 7-8
SETCOLOR 8-2
SETDRAW 8-19
SETMOUSE 6-62
SETTIME 2-35
SGET 8-32
SGN(4-2
SHEL_ENVRN(11-85
SHEL_FIND(11-84
SHEL_GET(11-82
SHEL_PUT(11-83
SHEL_READ(11-80
SHEL_WRITE(11-81
SHL&(4-22
SHL(4-22
SHL|(4-22
SHOWM 6-63
SHR&(4-22
SHR(4-22
SHR|(4-22
SIN(4-9
SINGLE{}= 2-19
SINGLE{ 2-19
SINQ(4-9
SLPOKE 2-17
SOUND 6-68
SPACE$(5-8
SPC(5-8
SPOKE 2-17
SPRITE 8-30
SPUT 8-32
SQR(4-7
SSORT 2-29
STEP 7-14
STICK 6-64
STICK(6-64
STOP 7-43
STORE 6-49
STR$(2-9
STRIG(6-64
STRING$(5-8
SUB 4-15, 7-24
SUB(4-17
SUCC(4-16, 5-4
SWAP 2-27
SWAP(4-26
SYSTEM 7-50

TAB(6-14
TAN(4-9
TEXT 8-29
THEN 7-3
TIME$ = 2-35
TIME$ 2-35
TIMER 2-36
TITLEW 9-26
TOPW 9-26
TOUCH 6-39
TRACE$ 7-53
TRIM$(5-5
TROFF 7-51
TRON 7-51, 7-53
TRUE 2-34
TRUNC(4-4
TYPE(2-7

UNTIL 7-16
UPPER$(5-9
USING 6-9, 6-47

V: 2-23
VAL(2-11
VAL?(2-11
VARPTR(2-23
VAR 7-23, 7-25

VDIBASE 11-25
VDISYS 11-23
VOID 2-38
VQT_EXTENT(10-37
VQT_NAME(10-38
VST_LOAD_FONTS(10-35
VST_UNLOAD_FONTS(10-35
VSYNC 8-35
VTAB 6-14
V_CLRWK(10-33
V_CLSVWK(10-32
V_CLSWK(10-31
V_OPNVWK(10-32
V_OPNWK(10-31
V_UPDWK(10-33
V~H= 10-30
V~H 10-30

W: 10-3
WAVE 6-68
WEND 7-18
WHILE 7-18
WINDTAB 9-27
WINDTAB(9-27
WINDTAB()= 9-27
WIND_CALC(11-74
WIND_CLOSE(11-65
WIND_CREATE(11-63
WIND_DELETE(11-66
WIND_FIND(11-72
WIND_GET(11-67
WIND_OPEN(11-64
WIND_SET(11-70
WIND_UPDATE(11-73
WORD(4-27
WORK_OUT(10-26
WRITE 6-7, 6-47
W_HAND(9-25
W_INDEX(9-25

XBIOS(10-1
XOR(4-24
XOR 3-8
\ 3-2
^ 3-2
{} 2-19
{}= 2-19
~ 2-38

GFA Systemtechnik GmbH
Heerdter Sandberg 30
D-4000 Düsseldorf 11
Telefon 02 11/58 80 11